NATO Science for Peace and Security Series
D: Information and Communication Security - Vol. 53

# Engineering Secure and Dependable Software Systems

Edited by
Alexander Pretschner
Peter Müller
Patrick Stöckle

IOS
Press

This publication is supported by: The NATO Science for Peace and Security Programme

NATO
OTAN

Almost all technical systems currently either interface with or are themselves largely software systems. Software systems must not harm their environment, but are also often vulnerable to security attacks with potentially serious economic, political, and physical consequences, so a better understanding of security and safety and improving the quality of complex software systems are crucial challenges for the functioning of society.

This book presents lectures from the 2018 Marktoberdorf summer school *Engineering Secure and Dependable Software Systems*, an Advanced Study Institute of the NATO Science for Peace and Security Programme. The lectures give an overview of the state of the art in the construction and analysis of safe and secure systems. Starting from the logical and semantic foundations that enable reasoning about classical software systems, they extend to the development and verification of cyber-physical systems, which combine computational and physical components and have become pervasive in aerospace, automotive, industry automation, and consumer appliances. Safety and security have traditionally been considered separate topics, but several lectures in this summer school emphasize their commonalities and present analysis and construction techniques that apply to both.

The book will be of interest to all those working in the field of software systems, and cyber-physical systems in particular.

ISBN 978-1-61499-976-8

9 781614 999768

www.iospress.nl

ISBN 978-1-61499-976-8 (print)
ISBN 978-1-61499-977-5 (online)
ISSN 1874-6268 (print)
ISSN 1879-8292 (online)

# ENGINEERING SECURE AND DEPENDABLE SOFTWARE SYSTEMS

Sub-Series D: Information and Communication Security – Vol. 53
ISSN 1874-6268 (print)
ISSN 1879-8292 (online)

# Engineering Secure and Dependable Software Systems

Edited by

Alexander Pretschner

*Technische Universität München, Germany*

Peter Müller

*ETH Zürich, Switzerland*

and

Patrick Stöckle

*Technische Universität München, Germany*

IOS Press

Amsterdam • Berlin • Washington, DC

Published in cooperation with NATO Emerging Security Challenges Division

Proceedings of the NATO Advanced Study Institute on Engineering Secure and Dependable Software Systems
Marktoberdorf, Germany
31 July – 11 August, 2018

ISBN 978-1-61499-976-8 (print)
ISBN 978-1-61499-977-5 (online)

*Publisher*
IOS Press BV
Nieuwe Hemweg 6B
1013 BG Amsterdam
Netherlands
fax: +31 20 687 0019
e-mail: order@iospress.nl

*For book sales in the USA and Canada:*
IOS Press, Inc.
6751 Tepper Drive
Clifton, VA 20124
USA
Tel.: +1 703 830 6300
Fax: +1 703 830 2300
sales@iospress.com

PRINTED IN THE NETHERLANDS

# Preface

Almost all technical systems are nowadays in large part software systems themselves or interface with software systems. The ubiquity of software systems requires them not to harm their environment (safety); and at the same time makes them vulnerable to security attacks with potentially considerable economic, political, and physical damage. Better understanding security and safety; improving the general quality of complex software systems (cyber defense and new technologies to support the construction of information technology infrastructure) and the respective development processes and technologies is a crucial challenge for the functioning of society.

Security and safety, or reliability, both are essential facets of the trustworthiness of modern cyber-physical systems. Cyber-physical systems more and more tightly combine and coordinate subsystems consisting of both computational and physical elements. Such systems become indispensable in the domains of aerospace, automotive, industry automation, and consumer appliances. Protecting data within these systems from attacks by external attackers (security), and protecting the environment from the misbehavior of these systems (safety) are two subjects traditionally considered separate. However, a closer look reveals that the techniques for construction and analysis of software-based systems used in both security and safety are not necessarily fundamentally different.

Along these lines, the 2018 Marktoberdorf summer school on software engineering, the 39th of its kind, was concerned with engineering dependable software systems.

JOHN BARAS lectured on Formal Methods and Toolsuites for CPS Security, Safety and Verification.

In these five lectures he presented a general rigorous methodology for model-based systems engineering for cyber-physical systems (CPS), which uses in several key steps traditional and novel formal methods, and more specialized applications and deeper results in several areas.

He first presented a rigorous MBSE methodology, framework and tool-suites for CPS. Advances in Information Technology have enabled the design of complex engineered systems, with large number of heterogeneous components and capable of multiple complex functions, leading to the ubiquitous CPS. He then went on to present several methods addressing the key problem of motion planning and controls with safety and temporal constraints. He described the strengths and weaknesses of each method and provide explicit application examples, and emphasized the key challenge of developing an integrated framework for handling finite temporal and finite space tolerances (requirements, constraints). Finally, he presented several detailed vignettes in: Security and Trust in Networked Systems, Automotive CPS, Stable Path Routing in MANET, Composable and Assured Autonomy. Professor Baras introduced novel formal methods employing various partial ordered semirings, which he used for modeling and evaluating trust and for analyzing multi-metric problems on networks and graphs (multigraphs). He showed an example linking the MBSE methodology to hardware design for automotive controllers. He discussed the challenge and need for composable security and described some initial steps towards achieving this goal.

PATRICK COUSOT lectured on Abstract Interpretation.

He defined the structural rule-based and then fixpoint prefix trace semantics of a simple subset of C. He defined properties, in particular program properties and collecting semantics. Professor Cousot then formalized the abstraction and approximation of program properties and how a structural rule-based/fixpoint abstract semantics can be derived from the collecting semantics by calculational design. He showed that verification methods and program logics are (non-computable) abstractions of the program collecting semantics. Then, he introduced a few classical effectively computable abstractions of numerical properties of programs and discussed industrial static analysis applications. Finally, he introduced a few abstractions of symbolic properties of programs and discuss opened problems. He concluded by a list of formal methods that can be formalized as abstract interpretations.

VIJAY GANESH lectured on SAT and SMT Solvers: A Foundational Perspective.

Over the last two decades, software engineering (broadly construed to include verification, testing, analysis, synthesis, security) has witnessed a silent revolution in the form of SAT and SMT solvers. These tools are now integral to many analysis, synthesis, verification, and testing approaches. In his lectures, Professor Ganesh traced the important technical developments that underpin SAT and SMT solver technology, provided a detailed explanation of how they work, provided a proof complexity-theoretic view of solvers as proof systems, and described how users can get the most out of these powerful tools.

SUMIT GULWANI lectured on Programming by Examples.

Programming by Examples (PBE) involves synthesizing intended programs in an underlying domain-specific language (DSL) from example-based specifications. PBE is set to revolutionize the programming experience for both developers and end users. It can provide a 10-100x productivity increase for developers in some task domains, and can enable computer users, 99% of whom are non-programmers, to create small scripts to automate repetitive tasks. Two killer applications for this technology include data wrangling (an activity where data scientists today spend 80% time) and code refactoring (an activity where developers spend up to 40% time in a typical application migration scenario).

Dr. Gulwani discussed some principles behind designing useful DSLs for program synthesis. A key technical challenge in PBE is to search for programs in the underlying DSL that are consistent with the examples provided by the user. He discussed a divide-and-conquer based search paradigm that inductively reduces the problem of synthesizing a program with a certain top-level operator to simpler synthesis problems over its sub-programs by leveraging the operator's inverse semantics. Another challenge in PBE is to resolve the ambiguity in the example-based specification. He discussed two complementary approaches: (a) ranking techniques that can pick an intended program from among those that satisfy the specification, and (b) active-learning based user interaction models. The various concepts were illustrated using Flash Fill, FlashExtract, and FlashRelate—PBE technologies for data manipulation domains. The Microsoft PROSE SDK allows easy construction of such technologies. He did a hands-on exercise that involved building a synthesizer for a small part of the Flash Fill DSL using the PROSE framework.

ARIE GURFINKEL lectured on Algorithmic Logic-based Verification.
Developing an automated program verifier is an extremely difficult task. By its very nature, a verifier shares many of the complexities of an optimizing compiler and of an efficient automated theorem prover. From the compiler perspective, the issues include idiomatic syntax, parsing, intermediate representation, static analysis, and equivalence preserving program transformations. From the theorem proving perspective, the issues include verification logic, verification condition generation, synthesizes of sufficient inductive invariants, deciding satisfiability, interpolation, and consequence generation. Luckily, the cores of both compilers and theorem provers are well understood, well-defined, and readily available. In these lectures, Professor Gurfinkel examined how to build a state-of-the-art program verifier by re-using much of existing compilers and SMT-solvers. The lectures were based on the SeaHorn verification framework developed in collaboration between University of Waterloo and SRI International.

JOSEPH HALPERN lectured on "An Epistemic Foundation for Authentication Logics", "A knowledge-based analysis of the blockchain protocol," and finally "Knowledge and common knowledge in a distributed environment."

RUPAK MAJUMDAR lectured on Formal Methods for Software Controlling the Physical World.
A cyber-physical system (CPS) integrates computation and communication with the control of physical processes. CPSs are ubiquitous: for example, automotive control systems, medical devices, energy grids, etc. are all examples of CPSs. An important question is to come up with a cost effective yet high confidence design and verification methodology for these systems. This is a very broad problem. In this sequence of lectures, Professor Majumdar focused on one particular aspect: design of reactive controllers for CPSs from declarative specifications. In particular, he described the basis of a control design technique called abstraction-based control design (ABCD). In ABCD, one abstracts a continuous-state, time-sampled dynamical system into a finite 2-person game and uses reactive synthesis algorithms from finite-state games. A strategy extracted from the finite game can be refined to a controller for the original system. He described the basics of reactive synthesis, starting with basic algorithms for safety and reachability games, and going on to describing general algorithms using the mu-calculus, showing how abstraction works, and how ABCD can be optimized using multi-level abstrac-

tions. Along the way, he showed connections to foundational questions in logic and automata theory.

PETER MÜLLER lectured on Building Deductive Program Verifiers.

Deductive program verifiers attempt to construct a proof that a given program satisfies a given specification. Their implementations reflect the semantics of the programming language and the specification language, and often include elaborate proof search strategies to automate verification. Each of these components is intricate, which makes building a verifier from scratch complex and costly.

In this lecture series, Professor Müller presented an approach to build program verifiers as a sequence of translations from the source language and specification via intermediate languages down to a logic for which automatic solvers exist. This architecture reduces the overall complexity by dividing the verification process into simpler, well-defined tasks, and enables the reuse of essential elements of a program verifier such as parts of the proof search, specification inference, and counterexample generation. He introduced intermediate verification languages and demonstrate how they can encode interesting verification problems.

CATHERINE MEADOWS lectured on Maude-NPA and Formal Analysis of Cryptographic Protocols With Equational Theories.

Formal analysis of cryptographic protocols has been one of the most successful applications of formal methods to security. It has played a prominent part in the development and validation of security standards, shown most recently by use of formal methods in the analysis of TLS 1.3 at the invitation of the TLS 1.3 developers.

One long-standing problem in the analysis of cryptographic protocols is reasoning about cryptosystems that satisfy equational properties, such as Diffie-Hellman and exclusive-or. In these cases the equational properties are necessary both to understand potential vulnerabilities, and to correctly represent the actions of the protocol. However, such equational properties can be difficult to incorporate so that the analysis is both tractable and sound. This is especially the case when the properties include associative-commutative rules. The Maude-NRL Protocol Analyzer (Maude-NPA) is a tool that takes a systematic approach to reasoning about protocols that rely on such properties, and it has had a major influence on work in the field.

This course consisted of two parts: an introduction to formal cryptographic protocol analysis, with emphasis on reasoning about equational properties, and an in-depth presentation on the Maude-NPA cryptographic protocol analysis tool, showing the theoretical and heuristic approaches it applies to reasoning about equational properties.

ANNABELLE MCIVER lectured on Qualitative and quantitative information flow with applications to security.

In this series, she described how to use abstraction and refinement to reason about confidentiality properties in sequential programs. First, a simple model based on Kripke structures was used to give a qualitative semantics to determine whether secret information is leaked inadvertently; those ideas were then generalised to enable the quantitative analysis of how much any leaked information can be used by an adversary.

Throughout the lectures the emphasis was on reasoning rules and the comparison of programs (and specifications) using a refinement order which maintains both functional consistency as well as information flow properties. The approach was illustrated by a number of well-known case studies drawn from the security literature.

MARC POUZET lectured on Synchronous Programming of Cyber-physical Systems.

Synchronous programming has enjoyed great success in the design and implementation of the most critical control software, e.g., aircraft (fly-by-wire, engine control, braking), railways (on board control, interlocking), nuclear plants (control software). Many of these applications are developed with the language SCADE, founded on pioneering work in Lustre, with key additions from Esterel and Lucid Synchrone.

Lustre is a domain specific language for programming the block diagrams that abound in engineering disciplines. It formalizes components as synchronous functions over streams (infinite sequences) expressed by recursively defined sets of dataflow equations. Models can be simulated and formally verified before being automatically compiled into code guaranteed to execute in bounded time and memory. This idea of 'model based design', radical for the time, is today at the heart of widely used industrial tools like Simulink.

This lecture presented the foundations of synchronous programming. Professor Pouzet came back to the origin of Lustre, its links with other stream language and the interest of a synchronous interpretation. He presented its semantics and modular compilation to software and show how it can be specified and verified formally in an interactive theorem prover (Coq). The lecture explained how the subtle mix of stream equations and hierarchical automata was designed and its integration into SCADE. He showed how type theory can be applied to express several safety properties and ensure a compilation into statically scheduled code for execution in finite memory, and also how those type systems can be extended for applications with multiple execution rates. Finally, he described the design, semantics and implementation of Zélus, where discrete controllers can be composed with continuous-time models of a physical environment.

ALEXANDER PRETSCHNER lectured on Accountability.

Accountability is the property of a system to help answer questions regarding why specific events have happened. With accountability infrastructures in place, identified reasons can be used to improve systems and to assign blame. Accountability rests on two pillars, monitoring and causality analyses. In this series of lectures, he focused on causality analyses and respective underlying models and showed their wide applicability in computer science. He considered causality analyses including spectrum-based fault localization, Granger causality analysis, model-based diagnosis, and focused on SAT-based and ILP-based approaches to counterfactual reasoning on the grounds of Halpern and Pearls notion of actual causality inference. Professor Pretschner also discussed the provenance of causal models as fault trees, attack trees, and explicit acyclic equations.

We thank all the lecturers, the staff, and hosts in Marktoberdorf. Specifically, Traudl Fichtl was once again instrumental in organizing and running the school. She helped make the 2018 Marktoberdorf summer school a most rewarding experience, both academically and personally.

The Editors

This page intentionally left blank

# Contents

This page intentionally left blank

*Engineering Secure and Dependable Software Systems*
*A. Pretschner et al. (Eds.)*
*IOS Press, 2019*
*© 2019 The authors and IOS Press. All rights reserved.*
*doi:10.3233/978-1-61499-977-5-1*

# Formal Methods and Tool-Suites for CPS Security, Safety and Verification

JOHN S. BARAS
*Institute for Systems Research and*
*Department of Electrical and Computer Engineering,*
*University of Maryland College Park, USA,*
*ACCESS Centre, Royal Institute of Technology (KTH), Stockholm, Sweden,*
*Institute for Advanced Study, Technical University of Munich (TUM), Germany*

**Abstract.** We summarize the material presented in our five lectures at the 2018 Marktoberdorf International Summer Schools on Engineering Secure and Dependable Software Systems. In these five lectures we presented a general rigorous methodology for model-based systems engineering for cyber-physical systems, which uses in several key steps traditional and novel formal methods and more specialized applications and deeper results in several areas.

**Keywords.** CPS, MBSE, validation, verification, reachability, semirings, composable

## 1. Introduction

Advances in Information Technology [1] have enabled the design of complex engineered systems, with large number of heterogeneous components and capable of multiple complex functions, leading to the ubiquitous cyber-physical systems (CPS). These advances have, at the same time, increased the capabilities of such systems and have increased their complexity to such an extent that systematic design towards predictable performance is extremely challenging, if not infeasible with current methodologies and tools. These rapidly expanding advances create tremendous opportunities for novel software systems use as both system components as well as design-manufacturing-operation tools, and consequently the need for developing novel formal methods for testing-validation-verification. The need to address both the cyber and the physical components leads to a critical need for new formal models beyond the current ones.

We summarize the material presented in our five lectures at the 2018 Marktoberdorf International Summer Schools on Engineering Secure and Dependable Software Systems. In these five lectures we presented a general rigorous methodology for model-based systems engineering for cyber-physical systems, which uses in several key steps traditional and novel formal methods and more specialized applications and deeper results in several areas.

The presentations of our five lectures are available from: https://drive.google.com/drive/folders/1J6tWP5C7s7JTob_FS2S_IO-KFV2A_51U. We refer to the references provided with this summary, for the detailed technical description of the methods and results presented. The papers and presentations cited are available from http://dev-baras.pantheonsite.io (or from the publishers sites).

## 2. Model-Based Systems Engineering for Cyber-Physical Systems

In Lectures 1 and 2 we presented a rigorous Model-Based Systems Engineering (MBSE) methodology, framework and tool-suites for Cyber-Physical Systems (CPS) [2–4]. The methodology and framework we presented [2–4] is aimed at catalyzing the development and use of interoperable methods and tools. The fundamental components in this MBSE methodology and framework and the associated challenges are: Architectures, Integrated Modeling Hubs, Development of System Structure and Behavior Formal Models, Allocation of Behavior to Structure, Tradeoff Analysis and Design Space Exploration, Requirements Management, Testing-Validation- Verification [3]. We emphasized the importance of linking multiple physics and cyber models through metamodeling, the run-time interaction between design space exploration and system models, and the current lack of integrated modeling and testing of requirements via formal models of various kinds [2–4].

We first discussed the "two faces" of Information Technology (IT) impact on Engineering, following [1]. This presentation was used to frame the two boundaries of the problem of synthesizing complex systems in an integrated and systematic method. The first, which we call the "existence proof", is the way biological systems are synthesized following their genetic programming. The second is the current engineering achievement of synthesizing VLSI chips by first designing them using an integrated software tool-suite and then sending the program, that describes the design and manufacturing of the chip, to a foundry, where specialized machines read and understand the instructions of the program and produce the chip. The gap between these two boundaries is the subject of intense research in various technological fields and a major engineering challenge. We then described progress made since the appearance of [1] including the design and manufacturing of aircraft (e.g. Boeing 777 to Boeing 787), the emergence of CPS, the ubiquitous social networks over the Web, renewable energy and smart grids, fast and inexpensive human genome generation, autonomous and connected cars, cloud computing, Internet of Things, Industrial internet, Industrie 4.0, crowd sourcing and manufacturing, smart homes, smart cities, wireless and networked embedded systems, the emergence of a network immersed world.

Our research identified the following fundamental challenges for the modeling, design, synthesis and manufacturing of CPS:

- Framework for developing cross-domain *integrated modeling hubs* for CPS;
- Framework for linking these integrated modeling hubs with tradeoff analysis methods and tools for *design space exploration*;
- Framework of linking these integrated synthesis environments with *databases of modular component and process* (manufacturing) models, backwards compatible with legacy systems;
- Framework for translating textual requirements to mathematical representations as constraints, rules, metrics involving both logical and numerical variables, *allocation of specifications* to components, to enable automatic *traceability* and *verification*.

It is the last challenge that clearly identifies the need for development of various formal models for representation of requirements and for their validation and verification. It represents a rich new area for expanding the theme for foundamental and applied contributions of the *Marktoberdorf International Summer Schools on Engineering Secure and Dependable Software Systems*. Our MBSE methodology proposes such an integration via the use of various formal models for requirements ranging from timed automata to timed Petri-Nets and several others, and the integration of model checking, contract based design and automatic theorem proving [2–4].

Our MBSE methodology integrates SysML (as a system architectural language used to describe the system structure and behavior [11–13, 19]), with Modelica (for multi-physics modeling [14, 15]), with MATLAB (for control and signal processing component modeling), and with various meta-modeling tools, most importantly the Functional Mock-up Interface (FMI) standard [14, 15, 20]. Composability can be addressed either via formal methods such as contract-based design [2–4] or via the inherently composable models of port-Hamiltonian Systems [23]. Our methodology integrates the resulting modeling hubs with design exploration tools that employ multi-criteria optimization and constrained based reasoning in an integrated way [2–4, 16]. We described applications of our MBSE methodology to several important technological problems: power grids [5], autonomous cars [22], aerospace [35–37], energy efficient buildings [10, 21], sensor networks [9, 18], communication networks [24–26], smart manufacturing [17], robotics [8], unmanned air vehicles, health care [32–34], cyber-security [43, 44, 46, 47, 51], social networks [27], disease modeling and analysis [32–34].

We described the new fundamental challenges faced when we consider networked CPS [28–31] and when incorporating humans as elements of such complex systems, a subject of rapidly increasing importance in view of the “networked society”, the IoT, and the “interconnected coevolving sociotechnical networks” paradigms. This description included the three layer interacting co-evolving multigraph model that we have developed [28], which consist of the collaboration network, the information network and the communication network, represented by multi-graphs with nodes and links annotated with weights that can be multivariable numeric, Boolean and even rule- based. The important problems of understanding the impact of the various topologies on performance of distributed algorithms for inference and decision-making were discussed, including our results on small world graphs and expanded graphs [28–31].

We described a novel formal method to control the complexity of design space exploration by grouping questions about related design variables that leads to provably faster response to design queries by several orders of time scale [6, 7].

We closed Lectures 1 and 2 with a description of what is lacking, research challenges and future promising research directions.

## 3. Motion Planning and Controls with Safety and Temporal Constraints

In Lectures 3 and 4, we presented several methods addressing the key problem of motion planning and controls with safety and temporal constraints. This is another technical area that provides a rich set of challenges and opportunities for foundamental and applied contributions of the *Marktoberdorf International Summer Schools on Engineering Secure and Dependable Software Systems* Lectures 3 and 4 were organized in the four parts described below.

**Part I:** Reachable set based safety verification and control synthesis

- I.1 Reachable set based verification [35–37]
- I.2 Control synthesis using optimization [35–38]

**Part II:** Motion planning for temporal logics with finite time constraints

- II.1 Mixed integer optimization based method [38, 39]
- II.2 Timed automata based method [40]

**Part III:** Event Triggered Controller Synthesis for Dynamical Systems with Temporal Logic Constraints [41]

**Part IV:** Event Triggered Feedback Control for Signal Temporal Logic Tasks [42]

We described the strengths and weaknesses of each method and provided explicit application examples. We emphasized the key challenge of developing an integrated framework for handling finite temporal and finite space tolerances (requirements, constraints).

## 4. Security and Trust in Networked Systems, Automotive CPS, Stable Path Routing in MANET, Composable and Assured Autonomy

In Lecture 5 we present several detailed vignettes in: Security and Trust in Networked Systems, Automotive CPS, Stable Path Routing in MANET, Composable and Assured Autonomy. The lecture was organized in the four parts described below.

**Part I:** Security and Trust in Networks and Networked Systems [43–47, 50, 51]

**Part II:** Hardware Software Co-design for Automotive CPS using Architecture Analysis and Design Language [52]

**Part III:** Distributed Topology Control for Stable Path Routing in Multi-Hop Wireless Networks [26, 48, 49]

**Part IV:** Composable and Assured Autonomy

We introduced novel formal methods employing various partial ordered semirings, which we use for modeling and evaluating trust and for analyzing multi-metric problems on networks and graphs (multigraphs). We showed an example linking the MBSE methodology to hardware design for automotive controllers. We discussed the challenge and need for composable security and described some initial steps towards achieving this goal.

## References

[1] J. S. Baras, keynote lecture, inaugural White Symposium, Univ. of Maryland, 2003. http://www.isr.umd.edu/files/JSB_White_Symposium_2003.

[2] J. S. Baras, inaugural lecture of Tage Erlander Guest Professorship at KTH, Stockholm, 2014. https://www.youtube.com/watch?v=1Ubiue-nrCU, http://www.kth.se/en/ees/omskolan/organisation/centra/access/newsandevents/tageerlander-guest-professorship-2014-1.478484.

[3] J. S. Baras and M. A. Austin,"Development of a Framework for CPS Open Standards and Platforms" ISR Techn. Report 2014-02, Univ. of Maryland 2014. http://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=0ahUKEwjzpd3IxJbNAhXoA8AKHTggDacQFggdMAA&url=http%3A%2F%2Fdrum.lib.umd.edu%2Fbitstream%2F1903%2F15084%2F3%2FTR_2014-02.pdf&usg=AFQjCNHHAlgJwcuhd_gi26tX7Q5P_1E5qg&sig2=w3WwdwPVlxzgU2HZvnrEkw.

[4] Joint workshop, hosted by the LCCC Linnaeus Center of Lund University and the ACCESS Linnaeus Center of KTH, on MBSE, May 4-6, 2015, Lund University, Lund Sweden. https://www.lccc.lth.se/index.php?page=LCCC-ACCESS-2015-05, https://www.lccc.lth.se/index.php?page=LCCC-ACCESS-2015-05-Program.

[5] D. Spyropoulos and J. S. Baras, "Extending Design Capabilities of SysML with Trade-off Analysis: Electrical Microgid Case Study", Proc. Conf. on Systems Engineering Research (CSER13), pp. 108-117, 2013.

[6] Y. Zhou, S. Yang, and J. S. Baras, "Compositional Analysis of Dynamic Bayesian Networks and Applications to Complex Dynamic System Decomposition", Proceedings of the Conference on Systems Engineering Research (CSER13), pp. 167-176, Atlanta, GA, March 19-22, 2013.

[7] S. Yang, B. Wang, and J. S. Baras, "Interactive Tree Decomposition Tool for Reducing System Analysis Complexity", Proc. Conf. on Systems Engineering Research (CSER13), pp. 138 147, March 19-22, 2013.

[8] Y. Zhou and J. S. Baras, "CPS Modeling Integration Hub and Design Space Exploration with Applications to Microrobotics", Chapter in the Volume Control of Cyber-Physical Systems, D. C. Tarraf (ed.), Lecture Notes in Control and Information Sciences 449, pp. 23-42, Springer 2013.

[9] B. Wang and J. S. Baras, "HybridSim: A Modeling and Co-simulation Toolchain for Cyber-Physical Systems", Proc. 17th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications, pp. 33-40, Delft, Netherlands, Oct. 30 Nov. 1, 2013.

[10] D. R. Daily, "Trade-off Based Design and Implementation of Energy Efficiency Retrofits In Residential Homes", MS Thesis, MSSE Program, University of Maryland, College Park, MD, 2014.

[11] S. Balestrini-Robinson, D. F. Freeman and D. C. Browne, "An Object-oriented and Executable SysML Framework for Rapid Model Development", Procedia Computer Science, vol. 44, p. 424, 2015.

[12] No Magic Inc., "Cameo Systems Modeler", No Magic, [Online]. Available: https://www.nomagic.com/products/cameo-systems-modeler#intro.

[13] No Magic Inc., "Modeling SysML Diagrams", No Magic, [Online]. Available: https://docs.nomagic.com/display/SYSMLP182/Modeling+SysML+Diagrams.

[14] Modelica Association Project, "Functional Mock-up Interface for Model Exchange and Co-Simulation", 25 July 2014. [Online]. Available: https://svn.modelica.org/fmi/branches/public/specifications/v2.0/FMI_for_ModelExchange_and_CoSimulation_v2.0.pdf.

[15] C. Paredis and A. Reichwein, "SysML-Modelica Integration", Model-Based Systems Engineering Center, Georgia Tech, [Online]. Available: http://www.mbsec.gatech.edu/research/projects/active/sysml-modelica-integration.

[16] J. Hooker, Logic-Based Methods for Optimization: Combining Optimization and Constraint Satisfaction: Combining Optimization and Constraint Satisfaction, Wiley-Interscience, 2000.

[17] D. Nau, M. Ball, J. Baras, A. Chowdhury, E. Lin, J. Meyer, R. Rajamani, J. Splain and V. Trichur, "Generating and Evaluating Designs and Plans for Microwave Modules", AI for Engineering Design, Analysis and Manufacturing (AI-EDAM), Vol. 14, No. 4, pp. 289-304, September 2000.

[18] B. Wang and J. S. Baras, "Integrated Modeling and Simulation Framework for Wireless Sensor Networks", Proc. 21st IEEE Intern. Conf. on Collaboration Technologies and Infrastructures (WETICE 2012- CoMetS track), pp. 268-273, Toulouse, France, June, 2012.

[19] No Magic Inc., "Simulation of SysML models", No Magic Inc., [Online]. Available: https://docs.nomagic.com/display/CST190/Simulation+of+SysML+models

[20] Dassault Systemes, "6.10.5 FMU Export from Simulink/ FMU Import into Simulink: The FMI Kit for Simulink", in Dymola Dynamic Modeling Laboratory User Manual - Volume 2, 2016, pp. 339-343.

[21] K. A. Cawasji and J. S. Baras, "SysML Executable Model of an Energy-Efficient House and Trade-Off Analysis", Proceedings 2018 IEEE Intern. Symp. on Systems Engineering, Rome, Italy, Oct. 1-3, 2018.

[22] S. Bansal, F. Alimardani, and J. S. Baras, "Model-Based Systems Engineering Applied to the Trajectory Planning for Autonomous Vehicles", Proceedings 2018 IEEE Intern. Symp. on Systems Engineering, Rome, Italy, Oct. 1-3, 2018.

[23] A. Van Der Schaft and D. Jeltsema, Port-Hamiltonian Systems Theory: An Introductory Overview, Now Publishers, 2014.

[24] J. S. Baras, V. Tabatabaee, P. Purkayastha and K. Somasundaram, "Component Based Performance Modeling of Wireless Routing Protocols", Proceedings IEEE ICC 2009 Ad Hoc and Sensor Networking Symposium, pp.1-6, Dresden, Germany, June 14-18, 2009.

[25] E. Paraskevas and J. S. Baras, "Component Based Modeling of Routing Protocols for Mobile Ad Hoc

Networks", Proc. Conf. on Information Sciences and Systems, pp. 1-6, Baltimore, MD, March 18-20, 2015.

[26] K. Somasundaram, J. S. Baras, K. Jain and V. Tabatabaee , "Distributed Topology Control for Stable Path Routing in Multi-hop Wireless Networks", Proceedings 49th IEEE Conference on Decision and Control (CDC 2010), pp. 2342-2347, Atlanta, Georgia, December 15-17, 2010.

[27] P. Gao, H. Miao, J.S. Baras and J. Golbeck, "STAR: Semiring Trust Inference for Trust - Aware Social recommenders", Proc. 10th ACM Conf. on Recommender Systems, Boston, MA, USA, September15-19, 2016.

[28] J. S. Baras, "A Fresh Look at Network Science: Interdependent Multigraphs Models Inspired from Statistical Physics", Proc. 6th Intern. Symposium on Communication, Control and Signal Processing, Invited Session, pp. 497-500, Athens, Greece, May 21-23, 2014.

[29] J. S. Baras and P. Hovareshti, "Effects of Topology in Networked Systems: Stochastic Methods and Small Worlds", Proc. 47th IEEE Conference on Decision and Control, pp. 2973-2978, Dec. 2008.

[30] A. Menon and J. S. Baras, "Expander Families as Information Patterns for Distributed Control of Vehicle Platoons", Proceedings 3rd IFAC Workshop on Distributed Estimation and Control in Networked Systems (NecSys 2012), pp. 288-293, Santa Barbara, California, September 14-15, 2012.

[31] A. Menon, J. Baras, "A Distributed Learning Algorithm with Bit-valued Communications for Multi-agent Welfare Optimization", Proc. 52nd IEEE Conference on Decision and Control, pp. 2406-2411, Dec. 2013.

[32] C. R. Kyrtsos and J. S. Baras, "Studying the role of APOE in Alzheimer's Disease Pathogenesis using a Systems Biology Model", Journal of Bioinformatics and Computational Biology, Vol. 11, No. 5 (2013), pp. 1342003-1 to 1342003-20, 2013.

[33] C. Kyrtsos and J. S. Baras, "Modeling the Role of the Glymphatic Pathway and Cerebral Blood Vessel Properties in Alzheimer's Disease Pathogenesis", PLOS One Journal, pp. 1-20, October 8, 2015; 10(10):e0139574. doi: 10.1371/journal.pone.0139574. eCollection 2015.

[34] I. M. Katsipis and J. S. Baras, "A Model-Based System Engineering Framework for Healthcare Management with Application to Diabetes Mellitus", Proc. 26th Intern. Conference on Software & Systems Engineering and their Applications, Telecom ParsTech, Paris, May 2015.

[35] Y. Zhou and J. S. Baras, "Reachable Set Approach to Collision Avoidance for UAVs", Proceedings of 54th IEEE Conference on Decision and Control, Osaka, Japan, December 15-18, 2015.

[36] Y. Zhou, A. Raghavan and J. S. Baras, "Time Varying Control Set Design for UAV Collision Avoidance Using Reachable Tubes", Proceedings of 55th IEEE Conference on Decision and Control, Las Vegas, USA, 2016.

[37] Y. Zhou, J. Moschler, and J. S. Baras, "A System Engineering Approach to Collaborative Coordination of UASs in the NAS with Safety Guarantees", Proceedings of the 2013 Integrated Communications Navigation and Surveillance Conference (ICNS), pp. 1-12, Herndon, VA, April 8-10, 2014.

[38] D. Maity and J. S. Baras "Motion Planning in Dynamic Environment with Bounded Time Temporal Logic Specifications", Proceeding of the 23rd Mediterranean Conference on Control and Automoation (MED 2015), pp. 973-979, Torremolinos, Spain, June 16-19, 2015.

[39] Y. Zhou, D. Maity and J. S. Baras, "Optimal Mission Planner with Timed Temporal Logic Constraints", Proceedings of 2015 European Control Conference, Linz, Austria, July 15-17,2015.

[40] Y. Zhou, D. Maity, and J. S Baras. "Timed Automata Approach for Motion Planning Using Metric Interval Temporal Logic", Proceedings of 2016 European Control Conference, Aalborg Denmark, June 29 - July 1, 2016.

[41] D. Maity and J.S. Baras, "Event-Triggered Controller Synthesis for Dynamical Systems with Temporal Logic Constraints", Proceedings 2018 American Control Conference, Milwaukee, USA, June 2729, 2018.

[42] L. Lindemann, D. Maity, J. Baras, and D. Dimarogonas, " Event-Triggered Feedback Control for Signal Temporal Logic Tasks," Proceedings 58th IEEE Conference on Decision and Control, Dec. 2018

[43] J.S. Baras and G. Theodorakopoulos, Path Problems in Networks, Synthesis Lectures on Communication Networks, Morgan & Claypool Publishers, February 2010.

[44] G. Theodorakopoulos and J. S. Baras, "On Trust Models and Trust Evaluation Metrics for Ad-Hoc Networks", Journal of Selected Areas in Communications, Security in Wireless Ad-Hoc Networks, Vol. 24, Number 2, pp. 318-328, February 2006. [2007, IEEE Communications Society Leonard G. Abraham Prize]

[45] G. Theodorakopoulos and J. S. Baras, "Linear Iterations on Ordered Semirings for Trust Metric Com-

putation and Attack Resiliency Evaluation", Proc. 17th International Symposium on Mathematical Networks and Systems, pp. 509-514, Kyoto, Japan, July 24-28, 2006.

[46] K.K. Somasundaram and J.S. Baras, "Performance Improvements in Distributed Estimation and Fusion Induced by a Trusted Core", Proceedings of the 12th International Conference on Information Fusion-Fusion 2009, pp.1942-1949, Seattle, Washington, USA, July 6-9, 2009.

[47] I. Matei, T. Jiang and J. S. Baras, "A Trust Based Distributed Kalman Filtering Approach for Mode Estimation in Power Systems", Proceeding of the First Workshop on Secure Control Systems (SCS) as part of CPSWeek 2010, pp. 1-6, Stockholm, Sweden, April 12, 2010.

[48] K. Somasundaram and J. S. Baras, "Solving Multi-metric Network Problems: An Interplay Between Idempotent Semiring Rules", J. of Linear Algebra and Applications Special Issue on the occasion of 1st Montreal Workshop on Idempotent and Tropical Mathematics, Volume 435, Issue 7, pp. 14941512, 1 October 2011.

[49] K. K. Somasundaram and J. S. Baras, "Semiring Pruning for Information Dissemination in Mobile Ad Hoc Networks", Proceedings of The First International Conference on Networks & Communications (NetCoM -2009), pp. 319  325, Chennai, India, December 27-29, 2009.

[50] K. Somasundaram and J. S. Baras, "Path Optimization and Trusted Routing in MANET: An Interplay Between Ordered Semirings," Proceedings of The Second International Conference on Networks & Communications (NetCoM - 2010), pp. 88-98, Chennai, India, December 27-29, 2010.

[51] E. Paraskevas, T. Jiang, P. Purkayastha and J. S. Baras, "Trust-Aware Network Utility Optimization in Multihop Wireless Networks with Delay Constraints", Proceedings of the 24th Mediterranean Conference on Control and Automation, pp. 593-598, Athens, Greece, June 21-24, 2016.

[52] Y. Zhou, J. S. Baras, S. Wang, "Hardware Software Co-design for Automotive CPS using Architecture Analysis and Design Language", Proceedings of the 5th Analytic Virtual Integration of Cyber-Physical Systems Workshop (AVICPS 2014), Rome, Italy, December 2, 2014.

This page intentionally left blank

*Engineering Secure and Dependable Software Systems*
*A. Pretschner et al. (Eds.)*
*IOS Press, 2019*
*© 2019 The authors and IOS Press. All rights reserved.*
*doi:10.3233/978-1-61499-977-5-9*

# A Formal Introduction to Abstract Interpretation

Patrick COUSOT
*Courant Institute of mathematical Sciences, New York University*

**Abstract.** We introduce basic concepts of abstract interpretation using the example of arithmetic and boolean expression semantics, properties, verification, static analysis, and their formal calculational design.

**Keywords.** Semantics, Property, Verification, Proof method, Static analysis, Calculational design.

## 1. Introduction

Abstract interpretation [1,2,3] aims at formalizing reasonings on the semantics of programs and automating the inference of properties of such semantics. The very basic concepts of abstract interpretation are illustrated using arithmetic and boolean expressions. We define their syntax, semantics, properties, and formally design static analyses of expressions by calculus.

## 2. The rule of signs

The Indian mathematician and astronomer Brahmagupta (born c. 598, died after 665) was the first to give rules to compute with zero and invented the rule of signs [4, page 151]. Verses 18.30–35 of his *Brāhma-sphuṭ-a-siddhānta* state

> [The sum] of two positives is positive, of two negatives negative; of a positive and a negative [the sum] is their difference; if they are equal it is zero. The sum of a negative and zero is negative, [that] of a positive and zero positive, [and that] of two zeros zero.
>
> . . .
>
> A negative minus zero is negative, a positive [minus zero] positive; zero [minus zero] is zero. When a positive is to be subtracted from a negative or a negative from a positive, then it is to be added.
>
> The product of a negative and a positive is negative, of two negatives positive, and of positives positive; the product of zero and a negative, of zero and a positive, or of two zeros is zero.

A positive divided by a positive or a negative divided by a negative is positive; a zero divided by a zero is zero[1]; a positive divided by a negative is negative; a negative divided by a positive is [also] negative.

A negative or a positive divided by zero has that [zero] as its divisor, or zero divided by a negative or a positive [has that negative or positive as its divisor]. The square of a negative or of a positive is positive; [the square] of zero is zero.

Following the pseudo-evaluation idea of Peter Naur in compilation [5,6], Michel Sintzoff [7] postulates the sign analysis in the following way:

“$a \times a + b \times b$ yields always the object “pos” when $a$ and $b$ are the objects “pos” or “neg”, and when the valuation is defined as follows :

$$\begin{array}{ll}
\text{pos+pos} = \text{pos} & \text{pos} \times \text{pos} = \text{pos} \\
\text{pos+neg} = \text{pos,neg} & \text{pos} \times \text{neg} = \text{neg} \\
\text{neg+pos} = \text{pos,neg} & \text{neq} \times \text{pos} = \text{neg} \\
\text{neg+neg} = \text{neg} & \text{neg} \times \text{neg} = \text{pos} \\
\text{V(p+q)} = \text{V(p)+V(q)} & \text{V(p} \times \text{q)} = \text{V(p)} \times \text{V(q)}
\end{array}$$

$$\text{V(0)} = \text{V(1)} = \ldots = \text{pos}$$

$$\text{V(-1)} = \text{V(-2)} = \ldots = \text{neg}$$

The valuation of $a \times a + b \times b$ yields “pos” by the following computation :

$$\begin{array}{rlrl}
\text{V}(a) &= \text{pos,neg} & \text{V}(b) &= \text{pos,neg} \\
\text{V}(a \times a) &= \text{pos} \times \text{pos, neg} \times \text{neg} & \text{V}(b \times b) &= \text{pos} \times \text{pos, neg} \times \text{neg} \\
&= \text{pos,pos} = \text{pos} & &= \text{pos,pos} = \text{pos}
\end{array}$$

$$\text{V}(a \times a + b \times b) = \text{V}(a \times a)\text{+V}(b \times b) = \text{pos+pos} = \text{pos”}$$

Observe that V(0×-1) = V(0)×V(-1) = pos×neg = neg while V(0×-1) = V(0) = pos. The error follows from an unsound handling of the abstraction V(0) of 0 into pos. The correct rule should be neq × pos = neg,pos which is less precise than Brahmagupta’s rule of signs which singles 0 out.

Our objective is to show that such abstract interpretations of the semantics of expressions can be designed formally, without error, by machine-checkable calculational design.

## 3. Sign analysis of iterative programs

The rule of signs generalizes to programs. For example the sign of `x` in

```
x = 0; while (...) { x = x+1 }
```

(where the iteration condition `(...)` is ignored) can be determined as follows:

[1]This was Brahmagupta’s only error, $\frac{0}{0}$ is undefined.

- After zero iteration, when entering the loop, if ever, $\mathtt{x} = 0$;
- After one iteration, the sign of `x` is zero, 1 is positive, so the sum `x+1` of zero and positive is positive;
- For the basis, we have shown that after zero or one iteration, the sign of `x` is zero (at iteration 0) or positive (at iteration 1) that is positive after at most 1 iteration;
- For the induction step, if after at most $n \geqslant 0$ iterations, the sign of `x` is positive, then 1 is positive, so the sum `x+1` of positive and positive is positive after the next iteration;
- After at most $n+1$ iterations, `x` is positive (at the previous $n \geqslant 0$ iterations) or positive (at the $n+1$-th iteration) then `x` is positive after at most $n+1$ iterations;
- By recurrence on the number of iterations in the loop, `x` is positive in the loop.

## 4. Sign abstraction, informally

The abstraction is that you do not (always) need to know the absolute value of the arguments to know the sign of the result of an operation. This is sometimes precise (for example for the multiplication) but can be imprecise (for example the sign of the sum of a positive and a negative is unknown when ignoring the absolute value of the arguments). This is nevertheless useful in practice if you know what to do when you dont know the sign. For example, a compiler will not suppress the lower bound check when accessing an array with an index not known to be positive. Moreover, it is always possible to refine the abstraction to get more precise results. For example Brahmagupta states [4, page 151]

> [If] a smaller [positive] is to be subtracted from a larger positive, [the result] is positive; [if] a smaller negative from a larger negative, [the result] is negative; [if] a larger [negative or positive is to be subtracted] from a smaller [positive or negative, the algebraic sign of] their difference is reversednegative [becomes] positive and positive negative. ...

Knowing an interval of the possible values is more precise than just knowing the sign. Static interval analysis was introduced in [8,1].

Out objective is to formalize abstract interpretations of arithmetic expressions (like the rule of signs) and to show how the abstraction can be formally calculated out of the semantics of arithmetic expressions.

## 5. Syntax of expressions

Let us consider the language of expressions.

$$\begin{array}{ll}
\mathtt{x}, \mathtt{y}, \ldots \in \mathbb{V} & \text{variables } (\mathbb{V} \text{ not empty}) \\
\mathtt{A} \in \mathbb{A} ::= \mathtt{1} \mid \mathtt{x} \mid \mathtt{A}_1 \texttt{ - } \mathtt{A}_2 & \text{arithmetic expressions} \\
\mathtt{B} \in \mathbb{B} ::= \mathtt{A}_1 \texttt{ < } \mathtt{A}_2 \mid \mathtt{B}_1 \ \mathtt{nand} \ \mathtt{B}_2 & \text{boolean expressions} \\
\mathtt{E} \in \mathbb{E} ::= \mathtt{A} \mid \mathtt{B} & \text{expressions}
\end{array}$$

This context-free grammar [9] specifies sets of program syntactic entities, the set $\mathbb{V}$ of variables, $\mathbb{A}$ of arithmetic expressions, $\mathbb{B}$ of boolean expressions, and $\mathbb{E}$ of either arithmetic or boolean expressions. The mathematical variables $\mathtt{x}$, $\mathtt{y}$, $\mathtt{A}$, $\mathtt{B}$, and $\mathtt{E}$ denote arbitrary elements of these sets.

There syntax is defined by grammar rules such as $\mathtt{A} ::= \mathtt{1} \mid \mathtt{x} \mid \mathtt{A}_1 \mathtt{-} \mathtt{A}_2$ specifying that an arithmetic expression $\mathtt{A}$ is either the constant $\mathtt{1}$, a variable $\mathtt{x} \in \mathbb{V}$, or the difference $\mathtt{A}_1 \mathtt{-} \mathtt{A}_2$ of two arithmetic expressions $\mathtt{A}_1$ and $\mathtt{A}_2$. The set $\mathbb{V}$ of variables is left unspecified (usually it is an identifier starting with a letter followed by 0 or more letters or digits or special symbols like "_").

This grammar is ambiguous since $\mathtt{1 - 1 - 1}$ can either be understood as $(\mathtt{1 - 1}) \mathtt{-} \mathtt{1}$ or $\mathtt{1 -} (\mathtt{1 - 1})$. We choose the first alternative so the binary operator is left-associative. In boolean expressions, $\mathtt{nand}$ is left-associative and the arithmetic operators have priority over boolean operators (so $\mathtt{1-1<1-1-1}$ is $((\mathtt{1-1})\mathtt{<}((\mathtt{1-1})\mathtt{-1}))$ *i.e.* false $\mathsf{ff}$).

## 6. Structural definitions

Structural definitions are generalizations of recursive definitions on naturals. Assume that we want to define a total function $f \in \mathbb{E} \to S$ from the domain $\mathbb{E}$ to the codomain $S$, where $S$ is a set. A structural definition is a recursive definition of the form

- $f(\mathtt{1})$ and $f(\mathtt{x})$ are defined to be constants (so $f(\mathtt{1}) \triangleq c_{\mathtt{1}}$ and $f(\mathtt{x}) \triangleq c_{\mathtt{x}}$ where $c_{\mathtt{1}}, c_{\mathtt{x}} \in S$)[2];
- $f(\mathtt{A}_1 \mathtt{-} \mathtt{A}_2)$ and $f(\mathtt{A}_1 \mathtt{<} \mathtt{A}_2)$ are functions of $f(\mathtt{A}_1)$ and $f(\mathtt{A}_2)$ (so $f(\mathtt{A}_1 \mathtt{-} \mathtt{A}_2) \triangleq F_{\mathtt{-}}(f(\mathtt{A}_1), f(\mathtt{A}_2)))$, $f(\mathtt{A}_1 \mathtt{<} \mathtt{A}_2) \triangleq F_{\mathtt{<}}(f(\mathtt{A}_1), f(\mathtt{A}_2))$;
- $f(\mathtt{B}_1 \,\mathtt{nand}\, \mathtt{B}_2) \triangleq F_{\mathtt{nand}}(f(\mathtt{B}_1), f(\mathtt{B}_2))$ where $F_{\mathtt{-}}, F_{\mathtt{<}}, F_{\mathtt{nand}} \in S \times S \to S$.

For example $\mathbb{vars} \in \mathbb{E} \to \wp(\mathbb{V})$, the (possibly empty) set of variables $\mathbb{vars}[\![\mathtt{E}]\!] \in \wp(\mathbb{V})$ occurring in expression $\mathtt{E} \in \mathbb{E}$, is well-defined as

$$\begin{aligned}
\mathbb{vars}[\![\mathtt{1}]\!] &\triangleq \emptyset \\
\mathbb{vars}[\![\mathtt{x}]\!] &\triangleq \{\mathtt{x}\} \\
\mathbb{vars}[\![\mathtt{A}_1 \mathtt{-} \mathtt{A}_2]\!] &\triangleq \mathbb{vars}[\![\mathtt{A}_1]\!] \cup \mathbb{vars}[\![\mathtt{A}_2]\!] \\
\mathbb{vars}[\![\mathtt{A}_1 \mathtt{<} \mathtt{A}_2]\!] &\triangleq \mathbb{vars}[\![\mathtt{A}_1]\!] \cup \mathbb{vars}[\![\mathtt{A}_2]\!] \\
\mathbb{vars}[\![\mathtt{B}_1 \,\mathtt{nand}\, \mathtt{B}_2]\!] &\triangleq \mathbb{vars}[\![\mathtt{B}_1]\!] \cup \mathbb{vars}[\![\mathtt{B}_2]\!]
\end{aligned}$$

Structural definitions are the basis of denotational semantics introduced by Dana Scott and Christopher Strachey [10] (and called compositional in this context).

[2] $\triangleq$ is "is defined as".

## 7. Environments

In order to formally define the value of any expression *e.g.* `1 - 1 - 1` $= -1$, we need to know the value of variables occurring in expressions *e.g.* `x - 1` is 2 when $\mathtt{x} = 3$, `x - 1` is 42 when $\mathtt{x} = 43$, *etc.* We cannot enumerate the infinitely many cases $\ldots, \mathtt{x} = -1, \mathtt{x} = 0, \mathtt{x} = 1, \ldots$. So we use an environment $\rho \in \mathbb{E}\mathrm{v}$ where $\mathbb{E}\mathrm{v} \triangleq \mathbb{V} \to \mathbb{Z}$ that is a function $\rho$ mapping a variable `x` to its value $\rho(\mathtt{x})$ in the set $\mathbb{Z}$ of all mathematical integers. By reasoning on the function $\rho$ we can handle infinitely many cases at once. For example, in environment $\rho$, the value of `x - 1` is $\rho(\mathtt{x}) - 1$ where $\rho(\mathtt{x})$ is the value of variable `x`, 1 is the mathematical integer one and $-$ is the mathematical difference.

## 8. Structural semantics of expressions

Given an environment $\rho \in \mathbb{E}\mathrm{v} \triangleq \mathbb{V} \to \mathbb{Z}$ mapping variables $\mathtt{x} \in \mathbb{V}$ to their value $\rho(\mathtt{x}) \in \mathbb{Z}$, the value $\mathcal{A}[\![\mathtt{A}]\!]\rho \in \mathbb{Z}$ of an arithmetic expression $\mathtt{A} \in \mathbb{A}$ and $\mathcal{B}[\![\mathtt{B}]\!]\rho \in \mathbb{B}$ of a boolean expression $\mathtt{B} \in \mathbb{B}$ is structurally defined as follows.

$$\begin{aligned}
\mathcal{A}[\![\mathtt{1}]\!]\rho &\triangleq 1 \\
\mathcal{A}[\![\mathtt{x}]\!]\rho &\triangleq \rho(\mathtt{x}) \\
\mathcal{A}[\![\mathtt{A}_1\ \mathtt{-}\ \mathtt{A}_2]\!]\rho &\triangleq \mathcal{A}[\![\mathtt{A}_1]\!]\rho - \mathcal{A}[\![\mathtt{A}_2]\!]\rho \\
\mathcal{B}[\![\mathtt{A}_1\ \mathtt{<}\ \mathtt{A}_2]\!]\rho &\triangleq \mathcal{A}[\![\mathtt{A}_1]\!]\rho < \mathcal{A}[\![\mathtt{A}_2]\!]\rho \\
\mathcal{B}[\![\mathtt{B}_1\ \mathtt{nand}\ \mathtt{B}_2]\!]\rho &\triangleq \mathcal{B}[\![\mathtt{B}_1]\!]\rho \uparrow \mathcal{B}[\![\mathtt{B}_2]\!]\rho \\
\mathcal{S}[\![\mathtt{E}]\!] &\triangleq \mathcal{A}[\![\mathtt{E}]\!] \qquad \text{when} \qquad \mathtt{E} \in \mathbb{A} \\
\mathcal{S}[\![\mathtt{E}]\!] &\triangleq \mathcal{B}[\![\mathtt{E}]\!] \qquad \text{when} \qquad \mathtt{E} \in \mathbb{B}
\end{aligned} \tag{1}$$

`1`, `x`, `-`, `<`, `nand`, `A`, and `B` are syntactic objects *e.g.* strings of characters. 1, $\rho$, $-$, $<$, and $\uparrow$ are mathematical objects. The recursive definition is structural *i.e.* by induction on the syntax of expressions `E` (either arithmetic `A` or boolean `B`). The semantics of complex expressions $\mathcal{A}[\![\mathtt{A}]\!]$ or $\mathcal{B}[\![\mathtt{B}]\!]$ is defined in function of the semantics of simpler expressions until reaching basic cases $\mathcal{A}[\![\mathtt{1}]\!]\rho \triangleq 1$ and $\mathcal{A}[\![\mathtt{x}]\!]\rho \triangleq \rho(\mathtt{x})$ for which the value is constant. The "not and" or "nand" boolean operator $\uparrow$ is defined by the following truth table

| $a$ | tt | tt | ff | ff |
|---|---|---|---|---|
| $b$ | tt | ff | tt | ff |
| $a \uparrow b$ | ff | tt | tt | tt |

All other logical operators (negation , implication $\Rightarrow$, conjunction $\vee$, disjunction $\wedge$) can be defined in terms of $\uparrow$.

The functions $\mathcal{A}$ and $\mathcal{B}$ are total functions meaning that they are well-defined for all their arguments *i.e.* $\forall \mathtt{B} \in \mathbb{B}\ .\ \mathcal{B}[\![\mathtt{B}]\!] \in (\mathbb{V} \to \mathbb{Z}) \to \mathbb{B}$ and similarly for arithmetic expressions. The well-definedness property is therefore $P = \{\mathtt{B} \in \mathbb{B} \mid \mathcal{B}[\![\mathtt{B}]\!] \in (\mathbb{V} \to \mathbb{Z}) \to \mathbb{B}\})$. It's proof is by structural induction.

## 9. Proofs by structural induction

Proofs by structural induction are well suited for proving properties of structural definitions.

Proofs by structural induction generalize proofs by recurrence. To prove that a property $P$ holds for all expressions $\mathtt{E} \in \mathbb{E}$, we prove that the property holds for the basic cases `1` and `x`. Then assuming that the property holds for $\mathtt{A}_1$ and $\mathtt{A}_2$, we prove that it holds for $\mathtt{A}_1 \mathtt{-} \mathtt{A}_2$ and $\mathtt{A}_1 \mathtt{<} \mathtt{A}_2$. Moreover, assuming the property holds for boolean expressions $\mathtt{B}_1$ and $\mathtt{B}_2$, we prove that it also holds for $\mathtt{B}_1 \,\mathtt{nand}\, \mathtt{B}_2$. We conclude that $\mathbb{E} \subseteq P$.

## 10. Properties

### 10.1. Properties are sets

Properties (*e.g.* “to be an even integer”, “to be an odd natural”) can be understood as the set of mathematical objects that have this property (*e.g.* $2\mathbb{Z} \triangleq \{x \in \mathbb{Z} \mid \exists k \in \mathbb{Z} \,.\, x = 2k\}$ and $2\mathbb{N} + 1 = \{x \in \mathbb{N} \mid \exists k \in \mathbb{N} \,.\, x = 2k + 1\}$). So if $P$ is a property then $x \in P$ means $x$ has property $P$ while $x \notin P$ means $x$ does not have property $P$. For example $42 \in 2\mathbb{Z}$ but $43 \notin 2\mathbb{Z}$ while the factorial ! is well-defined for naturals but not integers so that $! \in \mathbb{N} \to \mathbb{N}$ and $! \notin \mathbb{Z} \to \mathbb{Z}$.

### 10.2. Implication, weaker and stronger properties

When considering properties as sets, logical implication is subset inclusion $\subseteq$. For example “to be greater that 42 implies to be positive” is $\{x \in \mathbb{Z} \mid x > 42\} \subseteq \{x \in \mathbb{Z} \mid x \geqslant 0\}$. If $P \subseteq Q$ then $P$ is said to be stronger/more precise than $Q$ and $Q$ is said to be weaker/less precise that $P$. Stronger/more precise properties are satisfied by less elements while weaker/less precise properties are satisfied by more elements. False $\mathtt{ff}$ *i.e.* $\emptyset$ is the strongest property while true $\mathtt{tt}$ *i.e.* $\mathbb{Z}$ is the weakest property of integers.

## 11. Semantic properties of expressions

By expression property we might mean a property of the syntax of the expression (such has `A` has 42 signs `-` more precisely `A` belongs to the set of expressions with 42 signs `-`). This is software metrics and metrology [11], of little interest to us.

Instead an expression property will be understood as a semantic property that is a property of the semantics of expressions.

The semantics $\mathcal{A}[\![\mathtt{A}]\!]$ of an expression `A` maps environments $\rho \in \mathbb{V} \to \mathbb{Z}$ to a values in $\mathbb{Z}$, $\mathcal{A}[\![\mathtt{A}]\!] \in (\mathbb{V} \to \mathbb{Z}) \to \mathbb{Z}$. Following Section **10**, a semantic property of an expression is a set of possible semantics hence belongs to $\wp((\mathbb{V} \to \mathbb{Z}) \to \mathbb{Z})$. If $P \in \wp((\mathbb{V} \to \mathbb{Z}) \to \mathbb{Z})$ is a semantic property, then $\mathcal{A}[\![\mathtt{A}]\!] \in P$ means that “`A` has property P”.

**Example 1** $P = \{b \mid \forall \rho \in \mathbb{V} \to \mathbb{Z} \,.\, b(\rho) = \mathtt{tt}\} \cup \{b \mid \forall \rho \in \mathbb{V} \to \mathbb{Z} \,.\, b(\rho) = \mathtt{ff}\}$ *is the semantic property of a boolean expression "to always evaluate to* $\mathtt{tt}$*" or "to always evaluate to* $\mathtt{ff}$*". For example* $\mathtt{x} * \mathtt{x} + 1 > 0$ *and* $\mathtt{x} * \mathtt{x} < 0$ *have this property but not* $\mathtt{x} * \mathtt{x} > 0$ *since* $\mathtt{x} * \mathtt{x} > 0$ *is sometimes true (when* $|\rho(\mathtt{x})| > 0$*) and sometimes false (when* $|\rho(\mathtt{x})| = 0$*). So* $\mathcal{B}[\![\mathtt{x} * \mathtt{x} + 1 > 0]\!] \in P$ *while* $\mathcal{B}[\![\mathtt{x} * \mathtt{x} > 0]\!] \notin P$*.* □

Notice that semantic properties $P$ of expressions are just a particular case of property of expressions *i.e.* the property $\{\mathtt{E} \in \mathbb{E} \mid \mathcal{S}[\![\mathtt{E}]\!] \in P\}$.

## 12. Collecting semantics of expressions

The collecting semantics of expressions is the strongest property of an expression.

$$\mathcal{C}[\![\mathtt{A}]\!] \triangleq \{\mathcal{A}[\![\mathtt{A}]\!]\} \in \wp((\mathbb{V} \to \mathbb{Z}) \to \mathbb{Z}) \tag{2}$$

Arithmetic expression $\mathtt{A}$ is said to have semantic property $P \in \wp((\mathbb{V} \to \mathbb{Z}) \to \mathbb{Z})$ if and only if $\mathcal{A}[\![\mathtt{A}]\!] \in P$ or equivalently $\mathcal{C}[\![\mathtt{A}]\!] \subseteq P$ so that $\mathcal{C}[\![\mathtt{A}]\!]$ is the strongest property of $\mathtt{A}$. The idea of collecting semantics was introduced in [1] (under the qualifier "static semantics") as a basis for proving the soundness of static analyzes.

The fact that $(\mathcal{A}[\![\mathtt{A}]\!] \in P) \Leftrightarrow (\mathcal{C}[\![\mathtt{A}]\!] \subseteq P)$ may suggest that the concept of collecting semantics is of poor interest. However, $x \in S \Leftrightarrow \{x\} \subseteq S$ is the basic idea for abstracting set theory into order/lattice theory [12] (which has the equivalent of $\subseteq$ but not of $\in$).

Similarly, the collecting semantics of boolean expressions is

$$\mathcal{C}[\![\mathtt{B}]\!] \triangleq \{\mathcal{B}[\![\mathtt{B}]\!]\} \in \wp((\mathbb{V} \to \mathbb{Z}) \to \mathbb{B})$$

Again the collecting semantics $\mathcal{C}[\![\mathtt{E}]\!]$ of expressions $\mathtt{E}$ is just a particular case of property of expressions *i.e.* the property $\{\mathtt{E}' \in \mathbb{E} \mid \mathcal{S}[\![\mathtt{E}']\!] \in \mathcal{C}[\![\mathtt{E}]\!]\}$ *i.e.* all expressions $\mathtt{E}'$ that have the same semantics as $\mathtt{E}$.

## 13. Proving semantic properties of expressions by structural induction

Semantic properties can be proved by structural induction on expressions. For basic cases the proof is $\mathcal{C}[\![\mathtt{1}]\!] \subseteq P$ and $\mathcal{C}[\![\mathtt{x}]\!] \subseteq P$. Assuming $\mathcal{C}[\![\mathtt{A}_1]\!] \subseteq P$ and $\mathcal{C}[\![\mathtt{A}_2]\!] \subseteq P$, we prove $\mathcal{C}[\![\mathtt{A}_1 \texttt{ - } \mathtt{A}_2]\!] \subseteq P$ and $\mathcal{C}[\![\mathtt{A}_1 \texttt{ < } \mathtt{A}_2]\!] \subseteq P$. Assuming $\mathcal{C}[\![\mathtt{B}_1]\!] \subseteq P$ and $\mathcal{C}[\![\mathtt{B}_2]\!] \subseteq P$, we prove that for $\mathcal{C}[\![\mathtt{B}_1 \,\mathtt{nand}\, \mathtt{B}_2]\!] \subseteq P$. By structural induction, we conclude that $\mathbb{E} \subseteq \{\mathtt{E} \in \mathbb{E} \mid \mathcal{C}[\![\mathtt{E}]\!] \subseteq P\}$ *i.e.* $\forall \mathtt{E} \in \mathbb{E} \,.\, \mathcal{C}[\![\mathtt{E}]\!] \subseteq P$.

By structural induction on expressions, we have (we use Church's lambda notation $\boldsymbol{\lambda}\, x \cdot e$ for the anonymous function mapping $x$ to the value of expression $e$ for $x$ [13] and $\boldsymbol{\lambda}\, x \in S \cdot e$ to mean that the parameter $x$ must belong to the set $S$)

$$\mathcal{C}[\![\mathtt{1}]\!] = \{\boldsymbol{\lambda}\, \rho \in (\mathbb{V} \to \mathbb{Z}) \cdot 1\}$$
$$\mathcal{C}[\![\mathtt{x}]\!] = \{\boldsymbol{\lambda}\, \rho \in (\mathbb{V} \to \mathbb{Z}) \cdot \rho(\mathtt{x})\}$$

$$\mathcal{C}[\![\texttt{A}_1 \texttt{ - } \texttt{A}_2]\!] = \{\boldsymbol{\lambda}\, \rho \in (\mathbb{V} \to \mathbb{Z}) \cdot f_1(\rho) - f_2(\rho) \mid f_1 \in \mathcal{C}[\![\texttt{A}_1]\!] \wedge f_2 \in \mathcal{C}[\![\texttt{A}_2]\!]\}$$

$$\mathcal{C}[\![\texttt{A}_1 \texttt{ < } \texttt{A}_2]\!] = \{\boldsymbol{\lambda}\, \rho \in (\mathbb{V} \to \mathbb{Z}) \cdot f_1(\rho) < f_2(\rho) \mid f_1 \in \mathcal{C}[\![\texttt{A}_1]\!] \wedge f_2 \in \mathcal{C}[\![\texttt{A}_2]\!]\}$$

$$\mathcal{C}[\![\texttt{B}_1 \texttt{ nand } \texttt{B}_2]\!] = \{\boldsymbol{\lambda}\, \rho \in (\mathbb{V} \to \mathbb{Z}) \cdot f_1(\rho) \uparrow f_2(\rho) \mid f_1 \in \mathcal{C}[\![\texttt{B}_1]\!] \wedge f_2 \in \mathcal{C}[\![\texttt{B}_2]\!]\} \qquad \square$$

For example $\mathcal{C}[\![\texttt{x - x}]\!] = \{\boldsymbol{\lambda}\, \rho \in (\mathbb{V} \to \mathbb{Z}) \cdot 0\}$.

## 14. Abstract sign properties

We let $\mathbb{P}^{\pm} \triangleq \{\bot_{\pm}, {<}0, {=}0, {>}0, {\leqslant}0, {\neq}0, {\geqslant}0, \top_{\pm}\}$ be the set of signs where ${<}0$ is "strictly negative", ${\geqslant}0$ is "positive or zero", *etc.*, ${=}0$ is "equal to zero", ${\neq}0$ is "different from zero" (*i.e.* "strictly negative or strictly positive"). $\top_{\pm}$ (top) is "unknown sign" (*i.e.* tt that is "negative, zero, or positive"), $\bot_{\pm}$ (bottom) is "no sign" (*i.e.* ff that is "neither negative, zero, nor positive") be the abstract properties of the sign abstract domain $\mathbb{P}^{\pm}$. For example, the sign of `x` at point $\ell$ of the conditional `if (0==1)` $^{\ell}$`x=1;` is $\bot_{\pm}$ since that point is unreachable.

The sign minus operation $-_{\pm} \in \mathbb{P}^{\pm} \times \mathbb{P}^{\pm} \to \mathbb{P}^{\pm}$ defines the sign $s_1 -_{\pm} s_2$ of `x - y` when `x` has sign $s_1$ and `y` has sign $s_2$.

| | | $s_2$ | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $s_1 -_{\pm} s_2$ | | $\bot_{\pm}$ | ${<}0$ | ${=}0$ | ${>}0$ | ${\leqslant}0$ | ${\neq}0$ | ${\geqslant}0$ | $\top_{\pm}$ |
| $s_1$ | $\bot_{\pm}$ | $\bot_{\pm}$ | $\bot_{\pm}$ | $\bot_{\pm}$ | $\bot_{\pm}$ | $\bot_{\pm}$ | $\bot_{\pm}$ | $\bot_{\pm}$ | $\bot_{\pm}$ |
| | ${<}0$ | $\bot_{\pm}$ | $\top_{\pm}$ | ${<}0$ | ${<}0$ | $\top_{\pm}$ | $\top_{\pm}$ | ${<}0$ | $\top_{\pm}$ |
| | ${=}0$ | $\bot_{\pm}$ | ${>}0$ | ${=}0$ | ${<}0$ | ${\geqslant}0$ | ${\neq}0$ | ${\leqslant}0$ | $\top_{\pm}$ |
| | ${>}0$ | $\bot_{\pm}$ | ${>}0$ | ${>}0$ | $\top_{\pm}$ | ${>}0$ | $\top_{\pm}$ | $\top_{\pm}$ | $\top_{\pm}$ |
| | ${\leqslant}0$ | $\bot_{\pm}$ | $\top_{\pm}$ | ${\leqslant}0$ | ${<}0$ | $\top_{\pm}$ | $\top_{\pm}$ | ${\leqslant}0$ | $\top_{\pm}$ |
| | ${\neq}0$ | $\bot_{\pm}$ | $\top_{\pm}$ | ${\neq}0$ | $\top_{\pm}$ | $\top_{\pm}$ | $\top_{\pm}$ | $\top_{\pm}$ | $\top_{\pm}$ |
| | ${\geqslant}0$ | $\bot_{\pm}$ | ${>}0$ | ${\geqslant}0$ | $\top_{\pm}$ | ${\geqslant}0$ | $\top_{\pm}$ | $\top_{\pm}$ | $\top_{\pm}$ |
| | $\top_{\pm}$ | $\bot_{\pm}$ | $\top_{\pm}$ | $\top_{\pm}$ | $\top_{\pm}$ | $\top_{\pm}$ | $\top_{\pm}$ | $\top_{\pm}$ | $\top_{\pm}$ |

The sign operator $-_{\pm}$ is imprecise for difference $(-)$. In contrast, the sign operator for multiplication of mathematical integers $(\times)$ is exact *i.e.* the sign of the result is exactly known from the sign of the parameters. The above sign minus operation $-_{\pm}$ is incorrect with machine integers because of overflows as found *e.g.* in the `int abs(int x) { return (x<0) ? -x : x; }` method in Java$^{\text{TM}}$ returning a wrong value for `Integer.Min_VALUE`.

## 15. Structural sign semantics of expressions

The sign of an expression depends upon the sign of its free variables. We represent the sign of variables by a sign environment $\overset{\pm}{\rho} \in \mathbb{V} \to \mathbb{P}^{\pm}$ such that $\overset{\pm}{\rho}(\texttt{x})$ is the sign of variable `x`.

The sign semantics $\mathcal{S}^{\pm}[\![\mathtt{A}]\!]\overset{\pm}{\rho}$ of an arithmetic expression $\mathtt{A}$ is the sign of the expression value when evaluated with variables which sign is given by the sign environment $\overset{\pm}{\rho}$. For example, if $\overset{\pm}{\rho}(\mathtt{x}) = {>}0$ and $\overset{\pm}{\rho}(\mathtt{y}) = {\leqslant}0$ then $\mathcal{S}^{\pm}[\![\mathtt{x}-\mathtt{y}]\!]\overset{\pm}{\rho} = {>}0$.

The structural sign semantics $\mathcal{S}^{\pm}[\![\mathtt{A}]\!] \in (\mathbb{V} \to \mathbb{P}^{\pm}) \to \mathbb{P}^{\pm}$ may be defined as follows.

$$\begin{aligned}
\mathcal{S}^{\pm}[\![\mathtt{1}]\!]\overset{\pm}{\rho} &= {>}0\\
\mathcal{S}^{\pm}[\![\mathtt{x}]\!]\overset{\pm}{\rho} &= \overset{\pm}{\rho}(\mathtt{x})\\
\mathcal{S}^{\pm}[\![\mathtt{A}_1 - \mathtt{A}_2]\!]\overset{\pm}{\rho} &= (\mathcal{S}^{\pm}[\![\mathtt{A}_1]\!]\overset{\pm}{\rho}) -_{\pm} (\mathcal{S}^{\pm}[\![\mathtt{A}_2]\!]\overset{\pm}{\rho})
\end{aligned}$$

To be more precise, if any of the variables has sign $\perp_{\pm}$, meaning "the expression is never evaluated" then the result is $\perp_{\pm}$, meaning "no result is ever returned". We say that signs are $\perp_{\pm}$-*strict* and define $\downarrow^{\pm}$ to enforce it[3].

$$\begin{aligned}
\downarrow^{\pm}[\overset{\pm}{\rho}]s &\triangleq (\![\, \exists \mathtt{y} \in \mathbb{V} \,.\, \overset{\pm}{\rho}(\mathtt{y}) = \perp_{\pm} \;?\; \perp_{\pm} \;:\; s \,]\!)\\
\mathcal{S}^{\pm}[\![\mathtt{1}]\!]\overset{\pm}{\rho} &= \downarrow^{\pm}[\overset{\pm}{\rho}]({>}0)\\
\mathcal{S}^{\pm}[\![\mathtt{x}]\!]\overset{\pm}{\rho} &= \downarrow^{\pm}[\overset{\pm}{\rho}](\overset{\pm}{\rho}(\mathtt{x}))\\
\mathcal{S}^{\pm}[\![\mathtt{A}_1 - \mathtt{A}_2]\!]\overset{\pm}{\rho} &= (\mathcal{S}^{\pm}[\![\mathtt{A}_1]\!]\overset{\pm}{\rho}) -_{\pm} (\mathcal{S}^{\pm}[\![\mathtt{A}_2]\!]\overset{\pm}{\rho})
\end{aligned} \tag{3}$$

By structural induction on $\mathtt{A}$, if $\exists \mathtt{x} \in \mathbb{V} \,.\, \overset{\pm}{\rho}(\mathtt{x}) = \perp_{\pm}$ then $\mathcal{S}^{\pm}[\![\mathtt{A}]\!]\overset{\pm}{\rho} = \perp_{\pm}$.

## 16. Soundness

We would like to prove that the sign semantics $\mathcal{S}^{\pm}[\![\mathtt{A}]\!]$ of an arithmetic expression $\mathtt{A}$ is a weaker property than the collecting semantics $\mathcal{C}[\![\mathtt{A}]\!]$. But $\mathcal{S}^{\pm}[\![\mathtt{A}]\!] \in (\mathbb{V} \to \mathbb{P}^{\pm}) \to \mathbb{P}^{\pm}$ while $\mathcal{C}[\![\mathtt{A}]\!] \in \wp((\mathbb{V} \to \mathbb{Z}) \to \mathbb{Z})$ and the concrete semantic properties in $\wp((\mathbb{V} \to \mathbb{Z}) \to \mathbb{Z})$ are hardly comparable to the abstract sign properties in $(\mathbb{V} \to \mathbb{P}^{\pm}) \to \mathbb{P}^{\pm}$.

The solution if to express abstract properties in $(\mathbb{V} \to \mathbb{P}^{\pm}) \to \mathbb{P}^{\pm}$ as a concrete property in $\wp((\mathbb{V} \to \mathbb{Z}) \to \mathbb{Z})$. For that purpose we will define a concretization function $\ddot{\gamma}_{\pm} \in ((\mathbb{V} \to \mathbb{P}^{\pm}) \to \mathbb{P}^{\pm}) \to (\wp((\mathbb{V} \to \mathbb{Z}) \to \mathbb{Z}))$ mapping an abstract property to an "equivalent" concrete property.

Then the concrete semantics implies the abstract semantics up to concretization in that for all arithmetic expressions $\mathtt{A}$,

$$\mathcal{C}[\![\mathtt{A}]\!] \subseteq \ddot{\gamma}_{\pm}(\mathcal{S}^{\pm}[\![\mathtt{A}]\!]).$$

## 17. Sign concretization

We define the sign concretization function $\ddot{\gamma}_{\pm}$ in several steps.

[3]The conditional expression is $(\![\, \mathtt{tt} \;?\; a \;:\; b \,]\!) = a$ and $(\![\, \mathtt{ff} \;?\; a \;:\; b \,]\!) = b$.

1. First we consider signs (in $\mathbb{P}^{\pm}$) as properties of integers (in $\wp(\mathbb{Z})$).

$$\begin{aligned} &\gamma_{\pm}(\bot_{\pm}) \triangleq \emptyset && \gamma_{\pm}(\leqslant 0) \triangleq \{z \in \mathbb{Z} \mid z \leqslant 0\} \\ &\gamma_{\pm}(<0) \triangleq \{z \in \mathbb{Z} \mid z < 0\} && \gamma_{\pm}(\neq 0) \triangleq \{z \in \mathbb{Z} \mid z \neq 0\} \\ &\gamma_{\pm}(=0) \triangleq \{0\} && \gamma_{\pm}(\geqslant 0) \triangleq \{z \in \mathbb{Z} \mid z \geqslant 0\} \\ &\gamma_{\pm}(>0) \triangleq \{z \in \mathbb{Z} \mid z > 0\} && \gamma_{\pm}(\top_{\pm}) \triangleq \mathbb{Z} \end{aligned} \tag{4}$$

2. Then we consider sign environments $\overset{\pm}{\rho} \in \mathbb{V} \to \mathbb{P}^{\pm}$ as properties of environments (in $\wp(\mathbb{V} \to \mathbb{Z})$). $\overset{\pm}{\rho}$ is the abstract property of all concrete environments $\rho$ such that for all variables $\mathtt{x}$, the sign of $\rho(\mathtt{x})$ is $\overset{\pm}{\rho}(\mathtt{x})$.

$$\dot{\gamma}_{\pm}(\overset{\pm}{\rho}) \triangleq \{\rho \in \mathbb{V} \to \mathbb{Z} \mid \forall \mathtt{x} \in \mathbb{V} \,.\, \rho(\mathtt{x}) \in \gamma_{\pm}(\overset{\pm}{\rho}(\mathtt{x}))\} \tag{5}$$

   Observe that if $\overset{\pm}{\rho}(\mathtt{x}) = \bot_{\pm}$ for some $\mathtt{x} \in \mathbb{V}$ then $\gamma_{\pm}(\overset{\pm}{\rho}(\mathtt{x})) = \emptyset$ so $\forall \mathtt{x} \in \mathbb{V} \,.\, \rho(\mathtt{x}) \in \gamma_{\pm}(\overset{\pm}{\rho}(\mathtt{x}))$ is false proving that $\dot{\gamma}_{\pm}(\overset{\pm}{\rho}) = \emptyset$. So the abstraction of false ($\emptyset \in \wp(\mathbb{V} \to \mathbb{Z})$) is any abstract environment $\overset{\pm}{\rho}$ with at least one variable $\mathtt{x}$ such that $\overset{\pm}{\rho}(\mathtt{x}) = \bot_{\pm}$.

3. Finally the concretization of abstract properties $\overline{P} \in (\mathbb{V} \to \mathbb{P}^{\pm}) \to \mathbb{P}^{\pm}$ is the concrete property $\ddot{\gamma}_{\pm}(\overline{P}) \in \wp((\mathbb{V} \to \mathbb{Z}) \to \mathbb{Z})$ defined as

$$\ddot{\gamma}_{\pm}(\overline{P}) \triangleq \{\mathcal{S} \in (\mathbb{V} \to \mathbb{Z}) \to \mathbb{Z} \mid \forall \overset{\pm}{\rho} \in \mathbb{V} \to \mathbb{P}^{\pm} \,.\, \forall \rho \in \dot{\gamma}_{\pm}(\overset{\pm}{\rho}) \,.\, \mathcal{S}(\rho) \in \gamma_{\pm}(\overline{P}(\overset{\pm}{\rho}))\} \tag{6}$$

   *i.e.* $\mathtt{A}$ has abstract property $\overline{P}$, that is $\mathcal{A}[\![\mathtt{A}]\!] \in \ddot{\gamma}_{\pm}(\overline{P})$, if and only if for all environments $\rho$ with signs $\overset{\pm}{\rho}$, the value $\mathcal{A}[\![\mathtt{A}]\!]\rho$ of arithmetic expression $\mathtt{A}$ has sign $\overline{P}(\overset{\pm}{\rho})$.

This is sound in that for all $\mathtt{A} \in \mathbb{A}$, $\mathcal{C}[\![\mathtt{A}]\!] \subseteq \ddot{\gamma}_{\pm}(\mathcal{S}^{\pm}[\![\mathtt{A}]\!])$.

Observe that in Section 2, the concretization of signs is defined as $\gamma(\text{pos}) = \{z \in \mathbb{Z} \mid z \geqslant 0\}$ and $\gamma(\text{neg}) = \{z \in \mathbb{Z} \mid z < 0\}$. A sound definition of the rule of signs for multiplication $\times$ with this interpretation of the rule of sign would have been $\text{pos} \times \text{neg} = \text{pos}, \text{neg}$ *i.e.* $\top_{\pm}$.

## 18. Sign lattice

Sign properties $\mathcal{P}^{\pm} \triangleq \{\gamma_{\pm}(s) \mid s \in \mathbb{P}^{\pm}\}$ of integers can be partially ordered by $\subseteq$ (*i.e.* implication) as represented by the Hasse diagram below where the nodes are the elements of $\mathcal{P}^{\pm}$ and there is a bottom-up arrow from $P \in \mathcal{P}^{\pm}$ to $P' \in \mathcal{P}^{\pm}$ when $P \subsetneq P'$ and no $Q \in \mathcal{P}^{\pm}$ such that $P \subsetneq Q \subsetneq P'$. So $P \subseteq Q$ if and only if there is a path from $P$ to $Q$ in the Hasse diagram.

The abstract signs $\mathbb{P}^{\pm}$ are an isomorphic representation of $\mathcal{P}^{\pm}$ as shown on the right, where the isomorphism is $\gamma_{\pm} \in \mathbb{P}^{\pm} \to \mathcal{P}^{\pm}$.

$$\mathcal{P}^{\pm} = \begin{tikzcd} & \mathbb{Z} \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\ \{z \mid z \leqslant 0\} \arrow[d, dash] \arrow[dr, dash] & \{z \mid z \neq 0\} \arrow[dl, dash] \arrow[dr, dash] & \{z \mid z \geqslant 0\} \arrow[dl, dash] \arrow[d, dash] \\ \{z \mid z < 0\} \arrow[dr, dash] & \{0\} \arrow[d, dash] & \{z \mid z > 0\} \arrow[dl, dash] \\ & \emptyset & \end{tikzcd} \qquad \mathbb{P}^{\pm} = \begin{tikzcd} & \top_{\pm} \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\ \leqslant 0 \arrow[d, dash] \arrow[dr, dash] & \neq 0 \arrow[dl, dash] \arrow[dr, dash] & \geqslant 0 \arrow[dl, dash] \arrow[d, dash] \\ <0 \arrow[dr, dash] & =0 \arrow[d, dash] & >0 \arrow[dl, dash] \\ & \bot_{\pm} & \end{tikzcd}$$

Therefore, the abstract sign properties are partially ordered by $\sqsubseteq_{\pm}$ defined by $s \sqsubseteq_{\pm} s'$ if and only if $\gamma_{\pm}(s) \subseteq \gamma_{\pm}(s')$. An algorithm for the inclusion $\sqsubseteq_{\pm}$ on $\mathbb{P}^{\pm}$ easily follows from this formal definition by case analysis.

**Remark 1** *Observe that* $-_{\pm}$ *is increasing in each of its parameters i.e. if* $s_1 \sqsubseteq_{\pm} s'_1$ *then* $s_1 -_{\pm} s_2 \sqsubseteq_{\pm} s'_1 -_{\pm} s_2$ *and* $s_2 \sqsubseteq_{\pm} s'_2$ *then* $s_1 -_{\pm} s_2 \sqsubseteq_{\pm} s_1 -_{\pm} s'_2$ *so that if* $s_1 \sqsubseteq_{\pm} s'_1$ *and* $s_2 \sqsubseteq_{\pm} s'_2$ *then* $s_1 -_{\pm} s_2 \sqsubseteq_{\pm} s'_1 -_{\pm} s'_2$. □

## 19. Sign abstraction, formally

An integer property like $2\mathbb{N}+1$ (odd naturals) can be over-approximated in $\mathbb{P}^{\pm}$ by sign properties $\{z \in \mathbb{Z} \mid z > 0\}$, $\{z \in \mathbb{Z} \mid z \geqslant 0\}$, and $\mathbb{Z}$. The best over-approximation of $2\mathbb{N}+1$ in $\mathbb{P}^{\pm}$ is $\{z \in \mathbb{Z} \mid z > 0\}$ since it is sound (in that $2\mathbb{N}+1 \subseteq \{z \in \mathbb{Z} \mid z > 0\}$) and the most precise/strongest (in that $\{z \in \mathbb{Z} \mid z > 0\} \subseteq \{z \in \mathbb{Z} \mid z \geqslant 0\} \subseteq \mathbb{Z}$).

More generally, the best over-approximation of any integer property $P \in \wp(\mathbb{Z})$ in $\mathbb{P}^{\pm}$ is given by the abstraction function

$$\begin{aligned} \alpha_{\pm}(P) \triangleq \llparenthesis\; & P \subseteq \emptyset \mathrel{?} \bot_{\pm} \\ \parallel\; & P \subseteq \{z \mid z < 0\} \mathrel{?} {<}0 \\ \parallel\; & P \subseteq \{0\} \mathrel{?} {=}0 \\ \parallel\; & P \subseteq \{z \mid z > 0\} \mathrel{?} {>}0 \\ \parallel\; & P \subseteq \{z \mid z \leqslant 0\} \mathrel{?} {\leqslant}0 \\ \parallel\; & P \subseteq \{z \mid z \neq 0\} \mathrel{?} {\neq}0 \\ \parallel\; & P \subseteq \{z \mid z \geqslant 0\} \mathrel{?} {\geqslant}0 \\ :\; & \top_{\pm} \rrparenthesis \end{aligned} \tag{7}$$

$\alpha_{\pm}(P)$ is the best over-approximation of $P \in \wp(\mathbb{Z})$ in $\mathbb{P}^{\pm}$ since

- $P \subseteq \gamma_{\pm}(\alpha_{\pm}(P))$ *i.e.* $\alpha_{\pm}(P)$ is an over-approximation/sound abstraction of $P$;
- if $\overline{P} \in \mathbb{P}^{\pm}$ and $P \subseteq \gamma_{\pm}(\overline{P})$ then $\alpha_{\pm}(P) \sqsubseteq_{\pm} \overline{P}$ *i.e.* $\alpha_{\pm}(P)$ is more precise than any other over-approximation/sound abstraction of $P$.

We have

$$s_1 \mathbin{\dot{-}_\pm} s_2 = \alpha_\pm(\{x - y \mid x \in \gamma_\pm(s_1) \wedge y \in \gamma_\pm(s_2)\}). \tag{8}$$

We can use the soundness requirement (8) as a definition of $s_1 \mathbin{\dot{-}_\pm} s_2 \triangleq \alpha_\pm(\{x-y \mid x \in \gamma_\pm(s_1) \wedge y \in \gamma_\pm(s_2)\})$ to design $\dot{-}_\pm$ by calculus. We have to consider all possible cases for $s_1$ and $s_2$. We show three cases $\top_\pm \mathbin{\dot{-}_\pm} \bot_\pm = \bot_\pm$, $<0 \mathbin{\dot{-}_\pm} \geqslant 0 = <0$, and $\geqslant 0 \mathbin{\dot{-}_\pm} \geqslant 0 = \top_\pm$.

$$\begin{aligned}
&- \quad \alpha_\pm(\{x - y \mid x \in \gamma_\pm(\top_\pm) \wedge y \in \gamma_\pm(\bot_\pm)\}) \\
&= \quad \alpha_\pm(\{x - y \mid x \in \mathbb{Z} \wedge y \in \emptyset\}) && \text{(def. } \gamma_\pm\text{)} \\
&= \quad \alpha_\pm(\emptyset) = \bot_\pm && \text{(def. } \alpha_\pm\text{)}
\end{aligned}$$

$$\begin{aligned}
&- \quad \alpha_\pm(\{x - y \mid x \in \gamma_\pm(<0) \wedge y \in \gamma_\pm(\geqslant 0)\}) \\
&= \quad \alpha_\pm(\{x - y \mid x < 0 \wedge y \geqslant 0\}) && \text{(def. } \gamma_\pm\text{)} \\
&= \quad \alpha_\pm(\{x \mid x < 0\}) = <0 && \text{(def. } \alpha_\pm\text{)}
\end{aligned}$$

$$\begin{aligned}
&- \quad \alpha_\pm(\{x - y \mid x \in \gamma_\pm(\geqslant 0) \wedge y \in \gamma_\pm(\geqslant 0)\}) \\
&= \quad \alpha_\pm(\{x - y \mid x \geqslant 0 \wedge y \geqslant 0\}) && \text{(def. } \gamma_\pm\text{)} \\
&= \quad \alpha_\pm(\mathbb{Z}) = \top_\pm && \text{(def. } \alpha_\pm\text{)}
\end{aligned}$$

The calculations can be formally certified by a proof verifier [14,15].

One can also consider all cases $s \in \mathbb{P}^\pm$ for $s_1 \mathbin{\dot{-}_\pm} s_2$ for given $s_1$, $s_2$ when needed, using a theorem prover to make the proof that $\{x - y \mid x \in \gamma_\pm(s_1) \wedge y \in \gamma_\pm(s_2)\} \subseteq \gamma_\pm(s)$, and returning $\top_\pm$ when the proof fails (*e.g.* times out). Among all possible answers $s$ for which the theorem prover could make the proof, the $\sqsubseteq_\pm$-minimal one is chosen, if any. Otherwise, an arbitrary $\sqsubseteq_\pm$-minimal one has to be selected. This is called predicate abstraction [16].

The finite join $\sqcup_\pm$ on $\mathbb{P}^\pm$ is defined such that $\sqcup_\pm\{s_i \mid i \in \Delta\} \triangleq \alpha_\pm(\bigcup\{\gamma_\pm(s_i) \mid i \in \Delta\})$. It follows that $s \sqcup_\pm s' = \{a \mid a \in \{<0, =0, >0\} \wedge (a \sqsubseteq_\pm s \vee a \sqsubseteq_\pm s')\}$ which directly yields an algorithm for computing $\sqcup_\pm$ on $\mathbb{P}^\pm$.

## *19.1. Abstraction of environment properties*

The best abstraction of an environment property $P \in \wp(\mathbb{V} \to \mathbb{Z})$ is

$$\dot{\alpha}_\pm(P) \triangleq \boldsymbol{\lambda}\, \mathtt{x} \in \mathbb{V} \cdot \alpha_\pm(\{\rho(\mathtt{x}) \mid \rho \in P\}) \tag{9}$$

*i.e.* for each variable $\mathtt{x}$ it is the sign of the set of values $\rho(\mathtt{x})$ in all environments $\rho$ satisfying $P$.

Observe that $\dot{\alpha}_\pm(P) \triangleq \dot{\bot}_\pm \triangleq \boldsymbol{\lambda}\, \mathtt{x} \in \mathbb{V} \cdot \bot_\pm$ while if $\overset{\pm}{\rho}(\mathtt{x}) = \bot_\pm$ then $\dot{\gamma}_\pm(\overset{\pm}{\rho}) = \emptyset$ so $\emptyset \in \wp(\mathbb{V} \to \mathbb{Z})$ has several possible abstractions in $\mathbb{P}^\pm$ but $\dot{\bot}_\pm$ is the pointwise $\dot{\sqsubseteq}_\pm$-smallest of them.

## *19.2. Abstraction of semantic properties*

The best abstraction of a semantic property $P \in \wp((\mathbb{V} \to \mathbb{Z}) \to \mathbb{Z})$ is

$$\ddot{\alpha}_\pm(P) \triangleq \boldsymbol{\lambda}\, \overset{\pm}{\rho} \in \mathbb{V} \to \mathbb{P}^\pm \cdot \alpha_\pm(\{\mathcal{S}(\rho) \mid \mathcal{S} \in P \wedge \rho \in \dot{\gamma}_\pm(\overset{\pm}{\rho})\}) \tag{10}$$

*i.e.* given a sign environment $\overset{\pm}{\rho}$, $\ddot{\alpha}_\pm(P)\overset{\pm}{\rho}$ is the sign of the possible results $\mathcal{S}(\rho)$ of the semantics $\mathcal{S} \in P$ with property $P$ for all environments $\rho$ with sign $\overset{\pm}{\rho}$.

## 20. Characteristic property of abstraction/concretization

The abstraction/concretization functions $\langle \alpha_\pm,\ \gamma_\pm \rangle$ are closely related in that for all $P \in \wp(\mathbb{Z})$ and $\overline{P} \in \mathbb{P}^\pm$, they satisfy

$$\alpha_\pm(P) \sqsubseteq_\pm \overline{P} \Leftrightarrow P \subseteq \gamma_\pm(\overline{P})$$

**Proof 1** *By definition* (4) *of $\gamma_\pm$ and* (7) *of $\alpha_\pm$, we observe that*

- *$\gamma_\pm$ is increasing i.e. if $s \sqsubseteq_\pm s'$ then $\gamma_\pm(s) \subseteq \gamma_\pm(s')$;*
- *$\alpha_\pm$ is increasing i.e. if $P \subseteq P'$ then $\alpha_\pm(P) \sqsubseteq_\pm \alpha_\pm(P')$;* (11)
- *if $\alpha_\pm(P) = s$ then $P \subseteq \gamma_\pm(s)$ so $\gamma_\pm \circ \alpha_\pm$ is extensive i.e. $P \subseteq \gamma_\pm \circ \alpha_\pm(P)$;* (12)
- *by case analysis, if $P = \gamma_\pm(s)$ then $\alpha_\pm(P) = s$ so $\alpha_\pm \circ \gamma_\pm$ is the identity hence reductive i.e. $\alpha_\pm \circ \gamma_\pm(s) \sqsubseteq_\pm s$ since $\sqsubseteq_\pm$ is reflexive.* (13)

*It follows that*

$$\begin{array}{ll}
\quad \alpha_\pm(P) \sqsubseteq_\pm \overline{P} & \\
\Rightarrow \gamma_\pm \circ \alpha_\pm(P) \subseteq \gamma_\pm(\overline{P}) & \wr \gamma_\pm \textit{ is increasing and def. function composition } \circ \wr \\
\Rightarrow P \subseteq \gamma_\pm(\overline{P}) & \wr \gamma_\pm \circ \alpha_\pm \textit{ is extensive and } \subseteq \textit{ transitive} \wr \\
\Rightarrow \alpha_\pm(P) \sqsubseteq_\pm \alpha_\pm \circ \gamma_\pm(\overline{P}) & \wr \alpha_\pm \textit{ is increasing and def. function composition } \circ \wr \\
\Rightarrow \alpha_\pm(P) \sqsubseteq_\pm \overline{P} & \wr \alpha_\pm \circ \gamma_\pm \textit{ is reductive and def. function composition } \circ \wr \quad \square
\end{array}$$

□

Similar results hold for $\langle \dot{\alpha}_\pm,\ \dot{\gamma}_\pm \rangle$, and $\langle \ddot{\alpha}_\pm,\ \ddot{\gamma}_\pm \rangle$, see (14).

## 21. Galois connection

The abstraction/concretization functions $\langle \alpha_\pm,\ \gamma_\pm \rangle$ satisfy $\forall P \in \wp(\mathbb{Z})\ .\ \forall \overline{P} \in \mathbb{P}^\pm\ .\ \alpha_\pm(P) \sqsubseteq_\pm \overline{P} \Leftrightarrow P \subseteq \gamma_\pm(\overline{P})$, which is the definition of a Galois connection, which we write $\langle \wp(\mathbb{Z}),\ \subseteq \rangle \xleftrightarrow[\alpha_\pm]{\gamma_\pm} \langle \mathbb{P}^\pm,\ \sqsubseteq_\pm \rangle$.

More generally,

**Definition 1 (Galois connection)** *a Galois connection $\langle \mathbb{P},\ \sqsubseteq \rangle \xleftrightarrow[\alpha]{\gamma} \langle \overline{\mathbb{P}},\ \overline{\sqsubseteq} \rangle$ is such that the concrete domain $\langle \mathbb{P}, \sqsubseteq \rangle$ and the abstract domain $\langle \overline{\mathbb{P}}, \overline{\sqsubseteq} \rangle$ are partial orders, $\alpha \in \mathbb{P} \to \overline{\mathbb{P}}$ is the abstraction function, $\gamma \in \overline{\mathbb{P}} \to \mathbb{P}$ is the concretization function, and $\forall P \in \mathbb{P}\ .\ \forall \overline{P} \in \overline{\mathbb{P}}\ .\ \alpha(P)\ \overline{\sqsubseteq}\ \overline{P} \Leftrightarrow P \sqsubseteq \gamma(\overline{P})$.* □

For example

$$\langle\wp(\mathbb{V}\to\mathbb{Z}),\ \subseteq\rangle \xleftrightarrow[\dot{\alpha}_\pm]{\dot{\gamma}_\pm} \langle\mathbb{V}\to\mathbb{P}^\pm,\ \dot{\sqsubseteq}_\pm\rangle \tag{14}$$

$$\langle\wp((\mathbb{V}\to\mathbb{Z})\to\mathbb{Z}),\ \subseteq\rangle \xleftrightarrow[\ddot{\alpha}_\pm]{\ddot{\gamma}_\pm} \langle(\mathbb{V}\to\mathbb{P}^\pm)\to\mathbb{P}^\pm,\ \dot{\sqsubseteq}_\pm\rangle.$$

**Proof 2** *For all* $P\in\wp(\mathbb{V}\to\mathbb{Z})$ *and* $\overset{\pm}{\rho}\in\mathbb{V}\to\mathbb{P}^\pm$*, we have*

$\dot{\alpha}_\pm(P)\ \dot{\sqsubseteq}_\pm\ \overset{\pm}{\rho}$

$\Leftrightarrow \forall \mathtt{x}\in\mathbb{V}\ .\ \dot{\alpha}_\pm(P)\mathtt{x}\sqsubseteq_\pm \overset{\pm}{\rho}(\mathtt{x})$ $\quad$ ≀*pointwise def. of* $\dot{\sqsubseteq}_\pm$∫

$\Leftrightarrow \forall \mathtt{x}\in\mathbb{V}\ .\ \alpha_\pm(\{\rho(\mathtt{x})\mid\rho\in P\})\sqsubseteq_\pm \overset{\pm}{\rho}(\mathtt{x})$ $\quad$ ≀*def.* (9) *of* $\dot{\alpha}_\pm$∫

$\Leftrightarrow \forall \mathtt{x}\in\mathbb{V}\ .\ \{\rho(\mathtt{x})\mid\rho\in P\}\subseteq\gamma_\pm(\overset{\pm}{\rho}(\mathtt{x}))$ $\quad$ ≀$\langle\wp(\mathbb{Z}),\ \subseteq\rangle \xleftrightarrow[\alpha_\pm]{\gamma_\pm}\langle\mathbb{P}^\pm,\ \sqsubseteq_\pm\rangle$∫

$\Leftrightarrow \forall \mathtt{x}\in\mathbb{V}\ .\ \forall\rho\in P\ .\ \rho(\mathtt{x})\in\gamma_\pm(\overset{\pm}{\rho}(\mathtt{x}))$ $\quad$ ≀*def.* $\in$∫

$\Leftrightarrow \forall\rho\in P\ .\ \forall \mathtt{x}\in\mathbb{V}\ .\ \rho(\mathtt{x})\in\gamma_\pm(\overset{\pm}{\rho}(\mathtt{x}))$ $\quad$ ≀*def.* $\forall$∫

$\Leftrightarrow P\subseteq\{\rho\in\mathbb{V}\to\mathbb{Z}\mid\forall\mathtt{x}\in\mathbb{V}\ .\ \rho(\mathtt{x})\in\gamma_\pm(\overset{\pm}{\rho}(\mathtt{x}))\}$ $\quad$ ≀*def.* $\subseteq$∫

$\Leftrightarrow P\subseteq\dot{\gamma}_\pm(\overset{\pm}{\rho})$ $\quad$ ≀*def.* (5) *of* $\dot{\gamma}_\pm$*, proving* $\langle\wp(\mathbb{V}\to\mathbb{Z}),\ \subseteq\rangle \xleftrightarrow[\dot{\alpha}_\pm]{\dot{\gamma}_\pm}\langle\mathbb{V}\to\mathbb{P}^\pm,\ \dot{\sqsubseteq}_\pm\rangle$∫

*For all* $P\in\wp((\mathbb{V}\to\mathbb{Z})\to\mathbb{Z})$ *and* $\overline{P}\in(\mathbb{V}\to\mathbb{P}^\pm)\to\mathbb{P}^\pm$*, we have*

$\ddot{\alpha}_\pm(P)\ \dot{\sqsubseteq}_\pm\ \overline{P}$

$\Leftrightarrow \forall\overset{\pm}{\rho}\in\mathbb{V}\to\mathbb{P}^\pm\ .\ \ddot{\alpha}_\pm(P)\overset{\pm}{\rho}\sqsubseteq_\pm\overline{P}(\overset{\pm}{\rho})$ $\quad$ ≀*pointwise def. of* $\dot{\sqsubseteq}_\pm$∫

$\Leftrightarrow \forall\overset{\pm}{\rho}\in\mathbb{V}\to\mathbb{P}^\pm\ .\ \alpha_\pm(\{\mathcal{S}(\rho)\mid\mathcal{S}\in P\wedge\rho\in\dot{\gamma}_\pm(\overset{\pm}{\rho})\})\sqsubseteq_\pm\overline{P}(\overset{\pm}{\rho})$ $\quad$ ≀*def.* (10) *of* $\ddot{\alpha}_\pm$∫

$\Leftrightarrow \forall\overset{\pm}{\rho}\in\mathbb{V}\to\mathbb{P}^\pm\ .\ \{\mathcal{S}(\rho)\mid\mathcal{S}\in P\wedge\rho\in\dot{\gamma}_\pm(\overset{\pm}{\rho})\}\subseteq\gamma_\pm(\overline{P}(\overset{\pm}{\rho}))$

≀$\langle\wp(\mathbb{Z}),\ \subseteq\rangle \xleftrightarrow[\alpha_\pm]{\gamma_\pm}\langle\mathbb{P}^\pm,\ \sqsubseteq_\pm\rangle$∫

$\Leftrightarrow \forall\overset{\pm}{\rho}\in\mathbb{V}\to\mathbb{P}^\pm\ .\ \forall\mathcal{S}\in P\ .\ \forall\rho\in\dot{\gamma}_\pm(\overset{\pm}{\rho})\ .\ \mathcal{S}(\rho)\in\gamma_\pm(\overline{P}(\overset{\pm}{\rho}))$ $\quad$ ≀*def.* $\subseteq$∫

$\Leftrightarrow \forall\mathcal{S}\in P\ .\ \forall\overset{\pm}{\rho}\in\mathbb{V}\to\mathbb{P}^\pm\ .\ \forall\rho\in\dot{\gamma}_\pm(\overset{\pm}{\rho})\ .\ \mathcal{S}(\rho)\in\gamma_\pm(\overline{P}(\overset{\pm}{\rho}))$ $\quad$ ≀*def.* $\forall$∫

$\Leftrightarrow \forall\mathcal{S}\in P\ .\ \mathcal{S}\in\ddot{\gamma}_\pm(\overline{P})$ $\quad$ ≀*def.* $\in$ *and* (6) *of* $\ddot{\gamma}_\pm$∫

$\Leftrightarrow P\subseteq\ddot{\gamma}_\pm(\overline{P})$

≀*def.* $\subseteq$*, proving* $\langle\wp((\mathbb{V}\to\mathbb{Z})\to\mathbb{Z}),\ \subseteq\rangle \xleftrightarrow[\ddot{\alpha}_\pm]{\ddot{\gamma}_\pm}\langle(\mathbb{V}\to\mathbb{P}^\pm)\to\mathbb{P}^\pm,\ \dot{\sqsubseteq}_\pm\rangle.$∫ $\square$

## 22. Calculational design of the sign semantics of expressions

The soundness requirement in Section **17** is that $\forall\mathtt{A}\in\mathbb{A}\ .\ \mathcal{C}[\![\mathtt{A}]\!]\subseteq\ddot{\gamma}_\pm(\mathcal{S}^\pm[\![\mathtt{A}]\!])$. By the Galois connection of (14), this is equivalent to $\ddot{\alpha}_\pm(\mathcal{C}[\![\mathtt{A}]\!])\ \dot{\sqsubseteq}_\pm\ \mathcal{S}^\pm[\![\mathtt{A}]\!]$. Therefore the sign semantics is a sign abstraction of the collecting semantics. It follows that we can design $\mathcal{S}^\pm[\![\mathtt{A}]\!]$ by calculus, calculating $\ddot{\alpha}_\pm(\mathcal{C}[\![\mathtt{A}]\!])$ using $\dot{\sqsubseteq}_\pm$-over-approximation to avoid all computations made in the concrete domain.

- We first consider the case when $\exists\mathtt{x}\in\mathbb{V}\ .\ \overset{\pm}{\rho}(\mathtt{x})=\bot_\pm$ so that $\dot{\gamma}_\pm(\overset{\pm}{\rho})=\emptyset$.

$$
\begin{aligned}
& \ddot{\alpha}_\pm(\mathcal{C}[\![\mathtt{A}]\!])\overset{\pm}{\rho} \\
= \;& \alpha_\pm(\{\mathcal{S}(\rho) \mid \mathcal{S} \in \mathcal{C}[\![\mathtt{A}]\!] \wedge \rho \in \dot{\gamma}_\pm(\overset{\pm}{\rho})\}) && \wr\text{def. (10) of } \ddot{\alpha}_\pm\wr \\
= \;& \alpha_\pm(\{\mathcal{A}[\![\mathtt{A}]\!](\rho) \mid \rho \in \dot{\gamma}_\pm(\overset{\pm}{\rho})\}) && \wr\text{def. (2) of } \mathcal{C}[\![\mathtt{A}]\!]\wr \\
= \;& \alpha_\pm(\emptyset) && \wr\exists \mathtt{x} \in \mathbb{V} \,.\, \overset{\pm}{\rho}(\mathtt{x}) = \bot_\pm \text{ so that } \dot{\gamma}_\pm(\overset{\pm}{\rho}) = \emptyset\wr \\
= \;& \bot_\pm && \wr\text{def. (7) of } \alpha_\pm\wr \\
\triangleq \;& \mathcal{S}^\pm[\![\mathtt{A}]\!]\overset{\pm}{\rho}
\end{aligned}
$$

$\wr$in accordance with (3) such that, $\exists \mathtt{x} \in \mathbb{V} \,.\, \overset{\pm}{\rho}(\mathtt{x}) = \bot_\pm$ implies $\mathcal{S}^\pm[\![\mathtt{A}]\!]\overset{\pm}{\rho} = \bot_\pm$.$\wr$

- Then we consider the case when $\forall \mathtt{x} \in \mathbb{V} \,.\, \overset{\pm}{\rho}(\mathtt{x}) \neq \bot_\pm$ so that $\dot{\gamma}_\pm(\overset{\pm}{\rho}) \neq \emptyset$.

We proceed by structural induction on `A`.

– For the basic case of a constant `1`, we just apply the definitions.

$$
\begin{aligned}
& \ddot{\alpha}_\pm(\mathcal{C}[\![\mathtt{1}]\!])\overset{\pm}{\rho} \\
= \;& \alpha_\pm(\{\mathcal{S}(\rho) \mid \mathcal{S} \in \mathcal{C}[\![\mathtt{1}]\!] \wedge \rho \in \dot{\gamma}_\pm(\overset{\pm}{\rho})\}) && \wr\text{def. (10) of } \ddot{\alpha}_\pm\wr \\
= \;& \alpha_\pm(\{\mathcal{A}[\![\mathtt{1}]\!](\rho) \mid \rho \in \dot{\gamma}_\pm(\overset{\pm}{\rho})\}) && \wr\text{def. (2) of } \mathcal{C}[\![\mathtt{1}]\!]\wr \\
= \;& \alpha_\pm(\{1\}) && \wr\dot{\gamma}_\pm(\overset{\pm}{\rho}) \text{ is not empty and def. (1) of } \mathcal{A}[\![\mathtt{1}]\!]\wr \\
= \;& {>}0 && \wr\text{def. (7) of } \alpha_\pm\wr \\
\triangleq \;& \mathcal{S}^\pm[\![\mathtt{1}]\!]\overset{\pm}{\rho} && \wr\text{in accordance with (3) when } \forall \mathtt{y} \in \mathbb{V} \,.\, \overset{\pm}{\rho}(\mathtt{y}) \neq \bot_\pm\wr
\end{aligned}
$$

– For the basic case of a variable `x`, we apply the definitions and then simplify.

$$
\begin{aligned}
& \ddot{\alpha}_\pm(\mathcal{C}[\![\mathtt{x}]\!])\overset{\pm}{\rho} \\
= \;& \alpha_\pm(\{\mathcal{S}(\rho) \mid \mathcal{S} \in \mathcal{C}[\![\mathtt{x}]\!] \wedge \rho \in \dot{\gamma}_\pm(\overset{\pm}{\rho})\}) && \wr\text{def. (10) of } \ddot{\alpha}_\pm\wr \\
= \;& \alpha_\pm(\{\mathcal{A}[\![\mathtt{x}]\!](\rho) \mid \rho \in \dot{\gamma}_\pm(\overset{\pm}{\rho})\}) && \wr\text{def. (2) of } \mathcal{C}[\![\mathtt{x}]\!]\wr \\
= \;& \alpha_\pm(\{\rho(\mathtt{x}) \mid \rho \in \dot{\gamma}_\pm(\overset{\pm}{\rho})\}) && \wr\text{def. (1) of } \mathcal{A}[\![\mathtt{x}]\!]\wr \\
= \;& \alpha_\pm(\{\rho(\mathtt{x}) \mid \forall \mathtt{y} \in \mathbb{V} \,.\, \rho(\mathtt{y}) \in \gamma_\pm(\overset{\pm}{\rho}(\mathtt{y}))\}) && \wr\text{def. (5) of } \dot{\gamma}_\pm\wr \\
= \;& \alpha_\pm(\{\rho(\mathtt{x}) \mid \rho(\mathtt{x}) \in \gamma_\pm(\overset{\pm}{\rho}(\mathtt{x}))\})
\end{aligned}
$$

$\wr$since $\gamma_\pm(\overset{\pm}{\rho}(\mathtt{y}))$ is not empty so for $\mathtt{y} \neq \mathtt{x}$, $\rho(\mathtt{y})$ can be chosen arbitrarily to satisfy $\rho(\mathtt{y}) \in \gamma_\pm(\overset{\pm}{\rho}(\mathtt{y}))\wr$

$$
\begin{aligned}
= \;& \alpha_\pm(\{x \mid x \in \gamma_\pm(\overset{\pm}{\rho}(\mathtt{x}))\}) && \wr\text{letting } x = \rho(\mathtt{x})\wr \\
= \;& \alpha_\pm(\gamma_\pm(\overset{\pm}{\rho}(\mathtt{x}))) && \wr\text{since } S = \{x \mid z \in S\} \text{ for any set } S\wr \\
= \;& \overset{\pm}{\rho}(\mathtt{x}) && \wr\text{by (13), } \alpha_\pm \circ \gamma_\pm \text{ is the identity}\wr \\
\triangleq \;& \mathcal{S}^\pm[\![\mathtt{x}]\!]\overset{\pm}{\rho} && \wr\text{in accordance with (3) when } \forall \mathtt{y} \in \mathbb{V} \,.\, \overset{\pm}{\rho}(\mathtt{y}) \neq \bot_\pm\wr
\end{aligned}
$$

– For the inductive case of $\mathtt{A}_1 - \mathtt{A}_2$, we assume, by structural induction hypothesis, that $\ddot{\alpha}_\pm(\mathcal{C}[\![\mathtt{A}_1]\!]) \mathrel{\dot{\sqsubseteq}_\pm} \mathcal{S}^\pm[\![\mathtt{A}_1]\!]$ and $\ddot{\alpha}_\pm(\mathcal{C}[\![\mathtt{A}_2]\!]) \mathrel{\dot{\sqsubseteq}_\pm} \mathcal{S}^\pm[\![\mathtt{A}_2]\!]$

$$\ddot{\alpha}_\pm(\mathcal{C}[\![\mathtt{A}_1 - \mathtt{A}_2]\!])\overset{\pm}{\rho}$$

$= \alpha_\pm(\{\mathcal{S}(\rho) \mid \mathcal{S} \in \mathcal{C}[\![\mathtt{A}_1 \text{ - } \mathtt{A}_2]\!] \wedge \rho \in \dot{\gamma}_\pm(\overset{\pm}{\rho})\})$ $\wr$def. (10) of $\ddot{\alpha}_\pm \wr$

$= \alpha_\pm(\{\mathcal{A}[\![\mathtt{A}_1 \text{ - } \mathtt{A}_2]\!](\rho) \mid \rho \in \dot{\gamma}_\pm(\overset{\pm}{\rho})\})$ $\wr$def. (2) of $\mathcal{C}[\![\mathtt{A}_1 \text{ - } \mathtt{A}_2]\!]\wr$

$= \alpha_\pm(\{\mathcal{A}[\![\mathtt{A}_1]\!](\rho) - \mathcal{A}[\![\mathtt{A}_2]\!](\rho) \mid \rho \in \dot{\gamma}_\pm(\overset{\pm}{\rho})\})$ $\wr$def. (1) of $\mathcal{A}\wr$

$\sqsubseteq_\pm \alpha_\pm(\{x - y \mid x \in \{\mathcal{A}[\![\mathtt{A}_1]\!](\rho') \mid \rho' \in \dot{\gamma}_\pm(\overset{\pm}{\rho})\} \wedge y \in \{\mathcal{A}[\![\mathtt{A}_2]\!](\rho'') \mid \rho'' \in \dot{\gamma}_\pm(\overset{\pm}{\rho})\}\}$

$\wr\{f(\rho) - g(\rho) \mid \rho \in R\} \subseteq \{x - y \mid x \in \{f(\rho') \mid \rho' \in R\} \wedge y \in \{g(\rho'') \mid \rho'' \in R\}\}$ and $\alpha_\pm$ is increasing by (11).

This over-approximation allows for $\mathtt{A}_1$ and $\mathtt{A}_2$ to be evaluated in the concrete with different environments $\rho'$ and $\rho''$ with the same sign of variables but possibly different values of variables. This accounts for the fact that the rule of signs does not take relationships between values of variables into account. For example the sign of $\mathtt{x} \text{ - } \mathtt{x}$ is not $=0$ in general.$\wr$

$\sqsubseteq_\pm \alpha_\pm(\{x - y \mid x \in \gamma_\pm(\alpha_\pm(\{\mathcal{A}[\![\mathtt{A}_1]\!](\rho) \mid \rho \in \dot{\gamma}_\pm(\overset{\pm}{\rho})\}) \wedge y \in \gamma_\pm(\alpha_\pm(\{\mathcal{A}[\![\mathtt{A}_2]\!](\rho) \mid \rho \in \dot{\gamma}_\pm(\overset{\pm}{\rho})\}))\})$

$\wr\{x - y \mid x \in P \wedge y \in Q\} \subseteq \{x - y \mid x \in \gamma_\pm(\alpha_\pm(P)) \wedge y \in \gamma_\pm(\alpha_\pm(Q))\}$ since $\gamma_\pm \circ \alpha_\pm$ is extensive by (12) and $\alpha_\pm$ is increasing by (11).

This over-approximation allows for the evaluation of the sign to be performed in the abstract with $\text{-}_\pm$ instead of the concrete.$\wr$

$= \alpha_\pm(\{\mathcal{A}[\![\mathtt{A}_1]\!](\rho) \mid \rho \in \dot{\gamma}_\pm(\overset{\pm}{\rho})\}) \text{ -}_\pm\, \alpha_\pm(\{\mathcal{A}[\![\mathtt{A}_2]\!](\rho) \mid \rho \in \dot{\gamma}_\pm(\overset{\pm}{\rho})\})$

$\wr s_1 \text{ -}_\pm s_2 = \alpha_\pm(\{x - y \mid x \in \gamma_\pm(s_1) \wedge y \in \gamma_\pm(s_2)\})$ by (8)$\wr$

$= \alpha_\pm(\{\mathcal{S}(\rho) \mid \mathcal{S} \in \mathcal{C}[\![\mathtt{A}_1]\!] \wedge \rho \in \dot{\gamma}_\pm(\overset{\pm}{\rho})\}) \text{ -}_\pm\, \alpha_\pm(\{\mathcal{S}(\rho) \mid \mathcal{S} \in \mathcal{C}[\![\mathtt{A}_2]\!] \wedge \rho \in \dot{\gamma}_\pm(\overset{\pm}{\rho})\})$ $\wr$def. (2) of $\mathcal{C}[\![]\!]\wr$

$= \ddot{\alpha}_\pm(\mathcal{C}[\![\mathtt{A}_1]\!])\overset{\pm}{\rho} \text{ -}_\pm\, \ddot{\alpha}_\pm(\mathcal{C}[\![\mathtt{A}_2]\!])\overset{\pm}{\rho}$ $\wr$def. (10) of $\ddot{\alpha}_\pm\wr$

$= \ddot{\alpha}_\pm(\mathcal{C}[\![\mathtt{A}_1]\!])\overset{\pm}{\rho} \text{ -}_\pm\, \ddot{\alpha}_\pm(\mathcal{C}[\![\mathtt{A}_2]\!])\overset{\pm}{\rho}$ $\wr$def. (10) of $\ddot{\alpha}_\pm\wr$

$\sqsubseteq_\pm (\mathcal{S}^\pm[\![\mathtt{A}_1]\!]\overset{\pm}{\rho}) \text{ -}_\pm\, (\mathcal{S}^\pm[\![\mathtt{A}_2]\!]\overset{\pm}{\rho})$

$\wr$induction hypothesis and $\text{-}_\pm$ is increasing in both parameters by Remark 1$\wr$

$\triangleq \mathcal{S}^\pm[\![\mathtt{A}_1 \text{ - } \mathtt{A}_2]\!]\overset{\pm}{\rho}$ $\wr$in accordance with (3) when $\forall \mathtt{y} \in \mathbb{V} \,.\, \overset{\pm}{\rho}(\mathtt{y}) \neq \bot_\pm\wr$ □

## 23. Calculational design of abstract interpretations

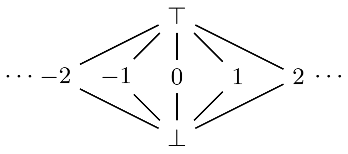
This concludes our formal design of the rule of signs for arithmetic expressions.

- We first define the semantics $\mathcal{A}[\![\mathtt{A}]\!]$ of arithmetic expressions $\mathtt{A}$ in (1);
- The strongest property of the semantics of arithmetic expressions $\mathtt{A}$ is their collecting semantics $\mathcal{C}[\![\mathtt{A}]\!]$ in (2);
- Among the semantic properties $\wp((\mathbb{V} \to \mathbb{Z}) \to \mathbb{Z})$ of arithmetic expressions, we select a subset of properties of interest *i.e.* the sign properties and choose a computer representation, as defined by the abstraction function $\ddot{\alpha}_\pm$ in (10), which is the lower adjoint of the Galois connection (14);
- The rule of sign $\mathcal{S}^\pm[\![\mathtt{A}]\!]$ is then formally derived by calculational design in Section **22** by over-approximating the best abstraction $\ddot{\alpha}_\pm(\mathcal{C}[\![\mathtt{A}]\!])$ of the collecting semantics $\mathcal{C}[\![\mathtt{A}]\!]$.

It follows that $\mathcal{S}^{\pm}[\![\mathtt{A}]\!]$ is sound by construction.

## 24. Classical finitary abstractions

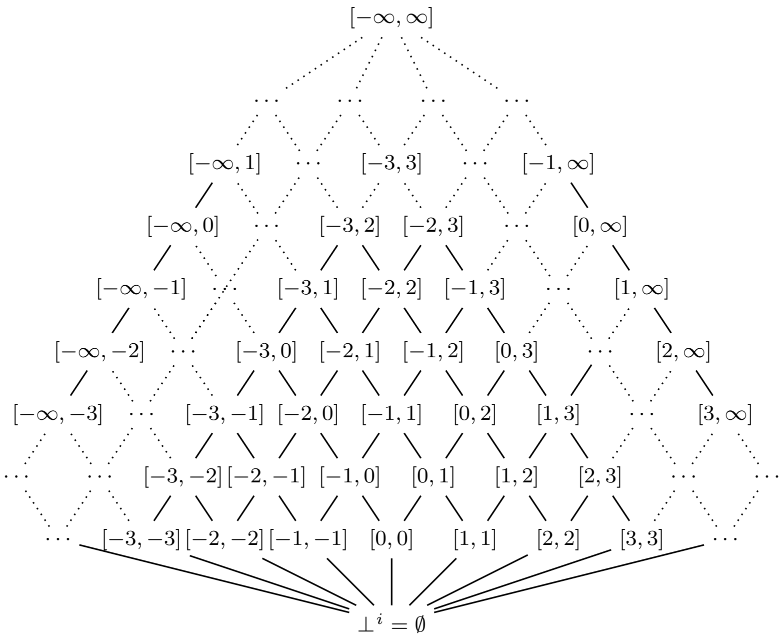
Other elementary examples are the parity analysis (which is correct with machine integers), [17] constancy analysis based on the lattice



such that $\gamma(\bot) = \emptyset$, $\gamma(i) = \{i\}$, $i \in \mathbb{Z}$, and $\gamma(\top) = \mathbb{Z}$.

## 25. Classical infinitary abstractions

As noticed by Brahmagupta, the sign analysis is not expressive enough to exactly determine the sign of expressions knowing the sign of its free variables. As shown by [18], interval analysis [8,1] will provide the desired answer. Interval analysis is based in the following lattice.


$[-\infty, \infty]$
$\cdots$ $\cdots$ $\cdots$ $\cdots$
$[-\infty, 1]$ $\cdots$ $[-3, 3]$ $\cdots$ $[-1, \infty]$
$[-\infty, 0]$ $\cdots$ $[-3, 2]$ $[-2, 3]$ $\cdots$ $[0, \infty]$
$[-\infty, -1]$ $\cdots$ $[-3, 1]$ $[-2, 2]$ $[-1, 3]$ $\cdots$ $[1, \infty]$
$[-\infty, -2]$ $\cdots$ $[-3, 0]$ $[-2, 1]$ $[-1, 2]$ $[0, 3]$ $\cdots$ $[2, \infty]$
$[-\infty, -3]$ $\cdots$ $[-3, -1]$ $[-2, 0]$ $[-1, 1]$ $[0, 2]$ $[1, 3]$ $\cdots$ $[3, \infty]$
$\cdots$ $\cdots$ $[-3, -2]$ $[-2, -1]$ $[-1, 0]$ $[0, 1]$ $[1, 2]$ $[2, 3]$ $\cdots$ $\cdots$
$\cdots$ $[-3, -3]$ $[-2, -2]$ $[-1, -1]$ $[0, 0]$ $[1, 1]$ $[2, 2]$ $[3, 3]$ $\cdots$
$\bot^i = \emptyset$

Because the lattice has infinite strictly increasing chains, the induction illustrated in Section **3** must be mechanized. This is the objective of widening and narrowing operators [8,1,19], see [20,21] for an introduction.

## 26. Conclusion

We have ilustrated the basics of abstract interpretation by defining the semantics of expressions, their properties, a proof method, and a sign analysis.

Instead of designing the rule of sign empirically and then proving its soundness, we used the soundness requirement as a guideline for designing the abstract sign semantics by calculus.

This sign analysis discovers an abstract property of an arithmetic expression by computing in the abstract only. This may involve some loss of precision, which was the case for the sign analysis.

The sign semantics is finite so it is an easily implementable static analysis. For infinite abstract domains, widening and narrowing operators are necessary.

## References

[1] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.

[2] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282. ACM Press, 1979.

[3] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 2(4):511–547, 1992.

[4] Kim Plofker. *Mathematics in India.* Princeton University Press, 2007.

[5] Peter Naur. The design of the GIER ALGOL compiler. *BIT Numerical Mathematics*, 3:124–140 and 145–166, June 1963.

[6] Peter Naur. Checking of operand types in ALGOL compilers. *BIT Numerical Mathematics*, 5:151–163, September 1965.

[7] Michel Sintzoff. Calculating properties of programs by valuations on specific models. In *Proceedings of ACM Conference on Proving Assertions About Programs*, pages 203–207. ACM, 1972.

[8] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[9] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956.

[10] Dana S. Scott and Christopher Strachey. Towards a mathematical semantics for computer languages. Technical report PRG-6, Oxford University Computer Laboratory, August 1971.

[11] International Organization for Standardization. Iso/iec 19761: Software engineering – cosmic: a functional size measurement method. March 2011.

[12] Garrett Birkhoff. *Lattice Theory.* American Mathematical Society, Colloquium publications, Volume XXV, third edition edition, 1973.

[13] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics. Series 2.*, 33(2):346–366, 1932.

[14] David Cachera and David Pichardie. Programmation d'un interprteur abstrait certifi en logique constructive. *Technique et Science Informatiques*, 30(4):381–408, 2011.

[15] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified C static analyzer. In *POPL*, pages 247–259. ACM, 2015.

[16] Susanne Graf and Hassen Sadi. Verifying invariants using theorem proving. In *CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 196–207. Springer, 1996.

[17] Gary A. Kildall. A unified approach to global program optimization. In *POPL*, pages 194–206. ACM Press, 1973.

[18] Roberto Giacobazzi and Francesco Ranzato. Completeness in abstract interpretation: A domain perspective. In *AMAST*, volume 1349 of *Lecture Notes in Computer Science*, pages 231–245. Springer, 1997.

[19] Patrick Cousot. Abstracting induction by extrapolation and interpolation. In *VMCAI*, volume 8931 of *Lecture Notes in Computer Science*, pages 19–42. Springer, 2015.

[20] P. Cousot and R. Cousot. *A gentle introduction to formal verification of computer systems by abstract interpretation*, pages 1–29. NATO Science Series III: Computer and Systems Sciences. IOS Press, 2010.

[21] Patrick Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.

This page intentionally left blank

*Engineering Secure and Dependable Software Systems*
*A. Pretschner et al. (Eds.)*
*IOS Press, 2019*
*© 2019 The authors and IOS Press. All rights reserved.*
*doi:10.3233/978-1-61499-977-5-29*

# SAT and SMT Solvers: A Foundational Perspective

Vijay Ganesh
*University of Waterloo, Ontario, Canada*

**Abstract.** Over the last two decades, we have witnessed a silent revolution in software engineering thanks in part to Boolean SAT and SMT solvers. These tools have made it significantly easier to design, build, and maintain myriad kinds of program analysis, testing, synthesis and verification systems. Further, they have enabled new software engineering methods that were otherwise deemed infeasible. In these lecture notes, we provide an introductory overview of SAT and SMT solvers, from both practical as well as theoretical points of view. Specifically, we describe in detail the architecture of the DPLL, CDCL, and SMT algorithms. Further, we provide theoretical understanding of these solvers through the lens of proof complexity. Finally, we showcase the power of machine learning techniques to fundamentally transform solver design.

**Keywords.** Boolean SAT solvers, first-order theories and SMT solvers, conflict-driven clause learning, proof complexity, machine learning for solvers

## 1. Introduction

Since the late 1990's, we have witnessed a dramatic transformation in software engineering research, wherein, many approaches to program analysis[8], synthesis[1], testing[8], security[36], model-based software engineering[19], and verification[7,9] are increasingly based on Boolean SATisfiability (SAT) and Satisfiability Modulo Theories (SMT) solvers[4,15,36,14,16,12]. The reason for this can primarily be attributed to the significant improvement in the efficiency and expressive power of these solvers. These enormous gains have brought renewed attention to an old idea that programs can and should be modeled mathematically, via appropriate fragments of logic, and that these models can be analyzed using solvers for the purpose of test generation or construction of proofs of correctness with respect to appropriate specifications. As software engineering/security methods and SAT/SMT solvers co-evolve, the demand for evermore powerful solvers continues unabated. In these introductory graduate-level notes, we provide an overview of the architecture of these solvers, the algorithms that power them, a theoretical understanding of these systems via proof complexity, as well as how machine learning can further enhance them, all with the aim of enabling users to more effectively leverage these versatile tools for their research needs.

**What is a Solver:** The typical definition of a solver is that it is a semi-decision procedure aimed at solving the satisfiability problem for a given fragment of mathematics. That is, solvers take as input well-formed formulas in said fragments, and decide whether they have solutions. Additionally, solvers may be required to generate solutions, if the input is satisfiable, and proofs of unsatisfiability, if the input is unsatisfiable. We say that a solver $S$ is sound, if whenever $S$ returns UNSAT, the input formula is indeed unsatisfiable. We say a solver $S$ is complete, if for any unsatisfiable formula that is an input to $S$, it returns UNSAT. We say a solver is terminating if it halts in finite time for all inputs.

A more modern view of a solver is that it is a sound proof system for a given fragment of mathematics[3,34]. (A solver may also be complete and terminating, depending on the fragment.) In these notes, we take the *solver is a proof system view*, since it is more amenable to a comprehensive complexity-theoretic analysis via proof complexity, as well as machine learning methods for optimal sequencing/selection of proof rules [1]. In addition to generating proofs or solutions, solvers are often required to support tactic languages and extensibility features (e.g., a programmatic interface), that make these tools versatile and easy to use. Finally, depending on the kind of mathematical fragment under consideration, the satisfiability problem can be in the complexity class $P$ (e.g., solving a system of linear equations over the reals), or $NP-complete$[10] (e.g., Boolean satisfiability), or undecidable (e.g., solving systems of Diophantine equations). All these requirements, applications, and complexity make solver design a very interesting and challenging field of study.

**DPLL SAT Solvers:** We start by describing the so-called DPLL SAT solving algorithm for Boolean logic, introduced in a series of papers from 1958 to 1962 by Davis, Putnam, Loveland, and Logemann[11]. The DPLL algorithm forms the basis for the CDCL SAT solving algorithm[27,29], and which in turn forms the foundation for most SMT solvers[12,14,31]. The DPLL SAT algorithm is a recursive backtracking search method that takes as input a Boolean formula $\phi(n)$ (in conjunctive normal form), and outputs SAT if the input has a solution and UNSAT otherwise.

Informally, the DPLL algorithm works as follows: the algorithm initially performs Boolean constraint propagation (BCP), i.e., *simplifies* the input with respect to clauses that have exactly one unassigned literal under the current partial assignment. If BCP alone is able to decide whether the input is SAT or UNSAT, the algorithm terminates and outputs the appropriate answer. Otherwise, the solver selects a variable and assigns it either true or false, and recursively calls itself. If the recursive call returns SAT, the algorithm terminates. Else, the recursive call returns UNSAT (referred to as a confict), meaning that the current partial assignment is not satisfying for the input formula. This causes the algorithm to backtrack, and assign the opposite value to the current decision variable and then recursively call itself again. This process continues until the algorithm either finds a satisfying assignment or determines that the input is unsatisfiable after it has searched through all possible assignments. We provide a more formal description of the DPLL SAT solving algorithm in Section 3.

**CDCL SAT Solvers:** Built on top of the DPLL method, the CDCL algorithm differs from it in many important ways such as clause learning[27], variable selection[29,24,23],

[1]While the *solvers as proof systems* view is very useful, one weakness of this approach is that proof systems typically don't deal with satisfiable instances. By contrast, solvers are semi-decision procedures and hence are required to handle satisfiable inputs as well. We don't consider this as problematic for two reasons: This view complements the *solvers as a search engine for solutions* view. They are both useful in deepening our understanding of solvers. Second, solvers do construct proofs of unsatisfiability for parts of the solution space that are empty, as they search for solutions for satisfiable instances.

value selection, restarts heuristics, clause deletion policies, and lazy data structures[4, 33]. Perhaps the most important idea that underlies the CDCL algorithm is clause learning. During its execution, when a CDCL solver determines that the current partial assignment to an input formula is unsatisfying (a conflicting assignment), a conflict analysis subroutine is invoked to examine why this is so. This kind of root cause analysis returns a clause, referred to as the learned or conflict clause, that is stored alongside the input formula (often in a database separate from the input formula). These learned clauses have two interesting properties: first, they are implied by the input formula, and perhaps more importantly, they prevent the solver from subsequently exploring not only the conflicting assignment, but also potentially exponentially many similar assignments thus cutting down the search space dramatically.

From a proof theoretic point of view, it can be shown that the DPLL method is polynomially equivalent to tree resolution proof system, while the CDCL method is polynomially equivalent to the far more powerful general resolution proof system[34]. This and other similar theoretical results enable us to establish the relative power of solver heuristics in a powerful way not possible otherwise.

**SMT Solvers:** Satisfiability Modulo Theories (SMT) solvers[31,12,14,16] are powerful algorithms designed to solve the satisfiability problem for first-order theories that are relevant in the context of program analysis, testing, security, and verification. They support a much richer input language than SAT solvers, e.g., first-order theories such as non-linear arithmetic over the reals and integers, theories over strings and regular expressions, bit-vectors, arrays, uninterpreted functions, datatypes, and floating-point arithmetic.

Broadly speaking there are two categories of SMT solvers, *eager* and *lazy*. Eager solvers are based on the idea of equisatisfiable reductions of input formulas in one theory into another (e.g., Bit-vectors to Boolean logic), and then invoking the appropriate solver on the resultant formula. The appeal of such an approach is that it leverages advances in solvers for one theory (e.g., SAT solvers) to efficiently solve formulas in another (e.g., bit-vectors). While this approach scales well for certain theories such as bit-vectors, for many other theories such as quantifier-free linear integer arithmetic this approach doesn't work well since reduction to Boolean logic may cause a considerable blowup in the size of the resultant formulas in terms of the formula from the input theory. Hence, researcher often use lazy SMT solvers for solving formulas from such theories.

By contrast to eager solvers, lazy SMT solvers are based on two key ideas, namely, abstraction-refinement and specialized decision procedures for first-order theories such as uninterpreted functions, arithmetic, strings, bit-vectors etc. At its core, a lazy SMT solver is a SAT solver extended with decision procedures for these first-order theories that enhance the SAT solver's propagation and conflict analysis subroutines. Informally, given a formula $\phi$ over first-order theories, the SMT solver first constructs a Boolean abstraction (an over-approximation) $\phi_b$ of $\phi$. If the abstraction is UNSAT, the solver returns UNSAT. Else, there is some satisfying assignment $A_b$ for $\phi_b$. The SMT solver then essentially verifies this assignment by constructing a conjunction of theory literals corresponding to literals in $A_b$, and calling the appropriate theory decision procedure on it. This process repeats until the solver correctly determines the satisfiability of the input formula.

**Perspective on Solvers via Proof Complexity and Machine Learning:** Instead of merely surveying existing literature on SAT and SMT solvers [4], we present in these notes the most important ideas that underpin solving algorithms via a *solvers as proof systems* perspective[34,3]. This perspective can be described briefly as follows: all solvers can be viewed as algorithms that implement proof systems aimed at proof search. Every proof rule in such systems may correspond to a set of subroutines in a solver (e.g., in the case of CDCL SAT solvers, the clause learning subroutine implements the general resolution rule, and the BCP implements the unit resolution rule). Additionally, solvers have sequencing and selection heuristics that attempt at optimal sequencing and selection of proof rules for a given class of instances (e.g., variable selection heuristics). The value of viewing *solvers as proof systems* is that it makes both the analysis and design of solver algorithms much more systematic than the ad-hoc application-driven approach followed today. More precisely, it brings two key benefits:

1. **Proof Complexity-theoretic Bounds on Solvers:** First, it enables us to leverage the vast literature on proof complexity to prove lower and upper bounds on solvers, that are otherwise hard to establish. Proof complexity also enables us to establish the power of certain heuristics, e.g., under appropriate assumptions CDCL solvers are polynomially equivalent to general resolution vs. DPLL solvers (which do not have clause learning) are only as powerful as tree-like resolution, a much weaker proof system.
2. **Machine Learning for Solver Design:** Second, a solver can be viewed as an implementation of a proof search algorithm for a proof system. That is, solver implementations consist of proof rules and methods for optimal sequencing and selection of these rules for a given input. This suggests that the problem of designing solver algorithms can be recast as coming up with appropriate proof rules and optimization procedures to adaptively sequence/select them for classes of instances. Given that current solvers produce copious amounts of data as they search for proofs and/or solutions, it further suggests that such optimization procedures may best be designed by leveraging machine learning algorithms[22]. An additional advantage of machine learning methods is that they can be adaptive, unlike ad-hoc heuristics [2].

In the rest of these notes, we not only discuss various solver algorithms, but also show how we can deepen our understanding of them by leveraging tools from proof complexity and machine learning, and thus enable the design of more efficient and adaptive solving algorithms going forward.

**Structure:** These notes are organized as follows. In Section 2 we introduce some relevant definitions and concepts from logic and complexity theory. In Section 3, we discuss DPLL SAT solvers laying the foundation for the subsequent sections. In Section 4, we discuss CDCL SAT solvers, with a sharp focus on clause learning, (machine-learning based) variable selection and restarts, and polynomial equivalence between CDCL SAT solvers and the general resolution proof system. In Section 5, we discuss SMT solvers with a sharp focus on their proof complexity. We conclude with reflection on key ideas underpinning SAT and SMT solver, their impact, and future directions.

[2]We stress here that the machine learning perspective of some of the sequencing and selection heuristics does not necessarily lend itself to any provable optimality results. Instead, the correspondence enables solver designers to move away from ad-hoc designs, and exploit the large amounts of data generated by solvers via appropriate machine learning methods that have already been developed

## 2. Preliminaries

Below we introduce definitions and concepts that are essential in the context of solvers. We assume the reader is familiar with basic ideas from mathematical logic and complexity theory. Terms that are specific to a certain class of solvers (e.g., branching heuristics in DPLL and CDCL solvers), will be introduced in the appropriate section.

### *2.1.* Propositional Logic

The constants in propositional logic (aka Boolean logic) are true and false respectively (also represented as 1 and 0 respectively). The notation and semantics of the Boolean operators $\neg$ (negation or complement), $\vee$ (disjunction), and $\wedge$ (conjunction) are standard. All formulas are constructed out of a finite set of Boolean variables, denoted as, $\{x_1, x_2, \ldots, x_n\}$. A Boolean variable or its negation is referred to as a literal. A clause is a disjunction of literals, written as $(x_1 \vee x_2 \vee \ldots \vee x_n)$. A formula is defined to be a conjunction of clauses, referred to as conjunctive normal form (CNF). We may sometimes refer to Boolean formulas as CNF formulas. It is easy to show that any Boolean function, can be translated into CNF via the Tseitin transformation, wherein, the CNF representation is only polynomially larger than the corresponding circuit in the number of variables[37].

The term *value* refers to elements of the set $\{true, false, u\}$[3], where $u$ represents *unknown*. An *assignment* $\mu$ is a map from variables of a formula to values $\{true, false, u\}$ (We may sometimes refer to an assignment to a variable as "setting a value to a variable"). We say an assignment $A$ is *complete* for a formula $\phi$ if $A$ is map such that all the variables of $\phi$ are assigned a value in true, false. Otherwise, we say that the assignment $A$ is *partial* for $\phi$. An assignment $A$ is empty for a formula $\phi$, if it is an empty map.

A variable evaluates to true (resp. false) under the assignment of the value 1 (resp. 0). The set of all assignments to the variables of a formula may sometimes be referred to as the *search space* of that formula. The search space of a formula can be represented as a binary tree, called the search or assignment tree, in the standard way. We say that clause is *unsatisfied* (or evaluates to false) under an assignment $\mu$, if all its literals are false under $\mu$. A clause is said to be *satisfied*, if at least one of its literals is true under $\mu$. A clause is unit if it has exactly one unassigned literal, under a partial assignment. The unassigned literal in a unit clause may sometimes be referred to as the unit literal of the said clause. We say a non-unit clause is unresolved under an assignment, if it is neither satisfied nor unsatisfied (i.e., evaluates to unknown and is not unit). The notion of evaluation of a formula to a value under an assignment is standard.

**The SAT Problem and SAT Solvers:** The satisfiability problem for Boolean logic can be stated as "Given a CNF formula, decide whether or not it is satisfiable". We say that formula is satisfiable if it has a solution, i.e., there exists an assignment to its variables such that the formula evaluates to true under this assignment. Otherwise, we say that the formula is unsatisfiable. It is well-known that this problem is NP-complete[10]. A SAT solver is a computer program designed to solve the Boolean satisfiability (or simply SAT) problem. All SAT solvers described here are sound, complete, and terminating decision procedures for propositional logic.

[3]We may choose to use 1 for true and 0 for false, for brevity.

## *2.2.* **First-order Logic and Theories**

We recall some standard definitions for first-order theories. Let $\mathscr{L}$ be a first-order signature (a list of constant, function, and predicate symbols). Given a set of $\mathscr{L}$-sentences $\mathscr{A}$ and an $\mathscr{L}$-sentence $B$ we write $\mathscr{A} \vdash B$ if every model of $\mathscr{A}$ is also a model of $B$. A *first order theory* (or simply a *theory*) is a set of $\mathscr{L}$-sentences that is consistent (that is, it has a model) and is closed under $\vdash$. The *decision problem* for a theory $T$ is the following: given a set $S$ of literals over $\mathscr{L}$, decide if there is a model $\mathscr{M}$ of $T$ such that $\mathscr{M} \models S$.

**The SMT Problem:** The Satisfiability Modulo Theories (SMT) problem is essentially the SAT problem for first-order theories. The *satisfiability problem* for $T$, also denoted $T$-SAT, is the following: given a quantifier-free formula $\mathscr{F}$ in $T$ in conjunctive normal form (CNF) (a $T$-formula), decide if there is a model $\mathscr{M}$ of $T$ such that $\mathscr{M} \models F$.

A simple example of a theory is E, the theory of equality. The signature of E contains a single predicate symbol $=$ and an infinite list of constant symbols. It is axiomatized by the standard axioms of equality (reflexivity, symmetry, and transitivity), and a sample sentence in E would be the formula $a \neq b \vee b \neq c \vee a = c$, which encodes the transitivity of equality between the constant symbols $a, b$, and $c$. Following the SMT literature, we will call terms from the theory (such as $a$ and $b$) *theory variables*, and the atoms derived from these terms (such as $a \neq b$ or $a = c$) will be called *theory literals* or just *literals*. We note that the decision problem for conjunctive fragment of E can be decided very efficiently [13]; in contrast, the satisfiability problem for E is easily seen to be *NP*-complete.

**Definition 2.1** (Unit Resolution)**.** Let $\mathscr{F}$ be a collection of clauses over an arbitrary theory $T$. A clause $C$ is derivable from $\mathscr{F}$ by *unit resolution* if there exists a resolution proof from $\mathscr{F}$ of $C$ such that in each application of the resolution rule, one of the clauses is a unit clause. If $C$ is derivable from $\mathscr{F}$ by unit resolution then we write $\mathscr{F} \vdash_1 C$. If $\mathscr{F} \vdash_1 \emptyset$ then we say $\mathscr{F}$ is *unit refutable*, otherwise it is *unit consistent*. We note that these definitions also apply to the Boolean case, where $T$ is simply Boolean logic.

### *2.2.1.* $\mathsf{Res}(T)$**: Resolution Modulo Theories**

Here we describe a new generalization of the general resolution proof system which captures reasoning modulo a first-order theory for constant symbols. We discuss two variants: the first, denoted $\mathsf{Res}(T)$, allows resolution to deduce in a single step any quantifier-free clause $C$ of literals occurring in the input formula such that $T \models C$. This is intended to model "standard" lazy SMT solvers [30] in which the solver is only allowed to reason about literals that already occurred in the input formula.

The second, more powerful variant (which we denote by $\mathsf{Res}^*(T)$) allows resolution to deduce any quantifier-free clause of literals $C$ such that $T \models C$, *even if* the new clause contains literals which do not occur in the input formula. This model is introduced to explore what would happen if a lazy SMT solver is allowed to introduce new literals from the theory. It is known that this can bring the complexity of a proof from exponential to polynomial, for example for diamond equalities $a_0 \neq a_n \wedge \bigwedge_{i=0}^{n-1}(a_i = b_i \wedge b_i = a_{i+1} \vee a_i = c_i \wedge c_i = a_{i+1})$ [5].

**Definition 2.2** ($\mathsf{Res}(T), \mathsf{Res}^*(T)$)**.** Let $T$ be a theory and let $\mathscr{F}$ be an quantifier-free CNF formula over $T$. The lines of a $\mathsf{Res}(T)$ ($\mathsf{Res}^*(T)$) proof are quantifier-free clauses of literals deduced from $\mathscr{F}$ and $T$ by the following derivation rules.

**Resolution.** $C \vee \ell, D \vee \overline{\ell} \vdash C \vee D$.
**Theory Derivation** ($\mathsf{Res}(T)$)**.** $\vdash C$ for any clause $C$ satisfying $T \models C$ and for which every variable in $C$ occurs in the input formula.
**Strong Theory Derivation** ($\mathsf{Res}^*(T)$)**.** $\vdash C$ for any clause $C$ satisfying $T \models C$.
A refutation of an unsatisfiable $\mathscr{F}$ is a proof in which the final line is the empty clause.

For the rest of these notes, any clause that is derived from a theory will be assumed to be quantifier-free. It is easy to see that both $\mathsf{Res}(T)$ and $\mathsf{Res}^*(T)$ are sound since the resolution rule and the Theory Derivation rule (for a non-trivial theory) are sound; completeness follows from a straightforward modification of the usual proof of resolution completeness (see, e.g. Jukna[20]).

Note that the clauses introduced by the theory derivations are arbitrary theorems of $T$; this means there is no direct information exchange between the resolution proof and the theory. It is enough to derive clauses in the theory derivation rules rather than arbitrary formulas since every axiom can be written in CNF form, and introduced as a sequence of clauses. The strong theory derivation rule can introduce new atoms (and thus new propositional variables to the Boolean abstraction of the initial formula) which might not have been present in the initial formula, and furthermore it seems that this ability to introduce new literals gives $\mathsf{Res}^*(T)$ extra power over general resolution.

## *2.3.* Comparing Proof Systems via Simulation

To compare proof systems over different languages we use the notion of an efficient simulation.

**Definition 2.3** (Polynomial Simulation)**.** A proof system A simulates a system B if there is a polynomial-time algorithm $R$ converting instances from the language of A to the language of B, and for every unsatisfiable formula $\mathscr{F}$, the shortest refutation proof of $\mathscr{F}$ in A is at most polynomially longer than the shortest refutation proof of $R(\mathscr{F})$ in B. We say two systems A and B are polynomially equivalent (or simply equivalent), if A simulates B and B simulates A.

## *2.4.* A Very Brief History of SAT and SMT Solvers

There are many surveys on the history of SAT and SMT solver. Perhaps the best is by John Franco in the Handbook of Satisfiability [4]. We briefly survey some of the key historical developments in solver research. As mentioned elsewhere, the DPLL algorithm was first developed by Davis, Putnam, Loveland, and Loeggemann[11]. The key concept of clause learning was developed by Marques-Silva and Sakallah [27] and the VSIDS heuristics was developed by Sharad Malik and his team [29]. The early work on SMT was focused primarily on decision procedures for individual theories and combination of decision procedure by pioneers Nelson and Oppen[32]. The most influential work on the modern architecture of SMT is by Nieuwenhuis, Oliveras, and Tinelli[31]. Perhaps the most influential SMT solver implementation is the Z3 solver by Bjorner and DeMoura[12].

**Algorithm 1** The DPLL SAT Solving Algorithm

```
1: function DPLL($\phi$, $\mu$)
2:     Input: A CNF formula $\phi$, and an initially empty assignment $\mu$
3:     Output: true (SAT) or false (UNSAT)
4:
5:     bcp_ret = Boolean_Constraint_Propagation($\phi$,$\mu$);
6:     if (bcp_ret == CONFLICT) then                ▷ If top-level conflict, return UNSAT
7:         dpll_ret = false;
8:     else
9:         if (all variables have been assigned) then        ▷ If solution found, return SAT
10:            dpll_ret = true;
11:        else  (Select decision variable $x$)              ▷ Heuristically select variable $x$
12:            dpll_ret = (DPLL($\phi$, $\mu : x$) || DPLL($\phi$, $\mu : \neg x$));      ▷ Recurse DPLL
13:    return dpll_ret;
```

## 3. The DPLL SAT Solving Algorithm

The pseudo code of the DPLL SAT solving algorithm[11], in an imperative-style language, is presented in Algorithm 1. The crucial steps in DPLL include the Boolean Constraint Propagation (BCP) subroutine on line 5, the variable selection heuristic on line 11, and the recursive calls on line 12.

### *3.1.* Boolean Constraint Propagation (BCP)

The BCP subroutine consists of repeated applications of the unit resolution rule until saturation, i.e., the subroutine has detected that the current assignment is satisfying or unsatisfying or there are no unit resolutions under the current partial assignment. The unit resolution rule is a special case of the general resolution rule, where at least one of the clauses input to the rule is unit. For example, consider the clauses $(x)$ and $(\neg x \vee \alpha)$, which when resolved result in derived clause $(\alpha)$ written as below [4]:

$$(x) \quad (\neg x \vee \alpha) \vdash (\alpha)$$

Repeated applications of the unit rule to an input formula amount to maintaining a queue of unit clauses, simplifying the formula with respect to the "current" unit clause (i.e., all occurrences of the current unit literal in the formula are assigned true, the complement of this unit literal are assigned false, and the clauses are appropriately simplified), adding any implied units to the unit clause queue, and repeating this process until this queue is empty. A variable $x$ that is assigned a value (alternatively, a variable whose value is set) as a result of applying BCP (one or more application of the unit resolution rule) is said to be *implied*.

BCP may return CONFLICT (i.e., the current partial assignment is unsatisfying for the input formula) or SAT (i.e., all variables have been assigned values true or false) or

[4]We choose to use the symbol $\vdash$ to denote a derivation step, with antecedents on the left side and consequent on the right.

unknown. If BCP returns CONFLICT at the top-level of the recursion (lines 6 and 7), then this means that the input formula is unsatisfiable (UNSAT). If, on the other hand, all the variables of the input formula have been assigned (line 9), then this means that the solver has found a satisfying assignment and it returns SAT (line 10). Else, it means that the BCP subroutine returns unknown, i.e., it cannot decide by itself whether the formula is SAT or UNSAT. This causes a variable selection heuristic to be invoked, that *select* an unassigned variable (line 11) (sometimes also referred to as a branching or decision variable) and recursively search for a satisfying assignment to the input formula (line 12).

## *3.2.* **Backtracking Search and DPLL**

Abstractly speaking, the DPLL algorithm implicitly constructs a binary search tree of assignments to the input formula $\phi$, and searches for a satisfying assignment in it. The nodes in this binary tree correspond to variables in $\phi$, and edges from each node denote assignment to the corresponding variable (typically, the left edge of a node is marked false and the right edge marked true). Leaf nodes of the tree are marked true (resp. false) depending on whether the path leading to a leaf corresponds to a satisfying assignment (resp. unsatisfying assignment) to $\phi$. As the DPLL algorithm performs BCP, selects an unassigned variable (line 11), and assigns a value to the branched variable (line 12), it correspondingly does a depth-first walk down the appropriate path in the binary search tree of assignments. At some level of the recursion, if the path terminates with a leaf marked false, the DPLL function correspondingly returns CONFLICT. This causes the solver to backtrack, and attempt the opposing value to the last branched variable (the second function call on line 12). It goes without saying that the recursion stack in runtime systems naturally supports backtracking over the search tree of assignments. This recursive search continues until either the solver determines that the input formula is SAT or UNSAT.

The notation $\mu : x$ (resp. $\mu : \neg x$) on line 12 of Algorithm 1 simply refer to adding the appropriate literal to the stack corresponding to the assignment $\mu$. The symbol $||$ corresponds to logical OR over two function calls. More precisely, it denotes an order over function calls, wherein, the left side recursive call is executed first, followed by the invocation of the right side only if the left side returns false.

## *3.3.* **Variable Selection Heuristics in Solvers**

Variable selection heuristics[5] are subroutines that take as input some fragment of the state of the solver (e.g., learned clauses), compute a total order over the variables of the input formula (line 11 in Algorithm 1), and output the highest ranked variable in this order. The selected variable is assigned a value and added to the current partial assignment, prior to the recursive calls to the DPLL solver on line 12. Solver researchers have understood for a long time that variable selection heuristics play a crucial role in the performance of solvers and a considerable amount of research has gone into their design and analysis. We will discuss modern variable selection heuristics in detail in Section 4.

[5]Variable selection heuristics are sometimes also referred to as branching, with the variable output by them referred to as branching variables. The term *decision heuristic* typically refers to the combination of variable and value selection heuristics. The literal returned by a decision heuristic is referred to simply as a *decision* or *decision literal*. The term *decision variable* refers to the variable that corresponds to a decision.

**Algorithm 2** The CDCL SAT Solving Algorithm

```
function CDCL($\phi$, $\mu$)
    Input: A CNF formula $\phi$, and an initially empty assignment trail $\mu$
    Output: SAT or UNSAT

    if (CONFLICT == Boolean_Constraint_Propagation($\phi$,$\mu$)) then
        return UNSAT;
    dl = 0;                                        ▷ : Initially, decision level dl is 0
    while all variables have NOT been assigned do                ▷ The search loop
        $x$ = DecisionHeuristic($\phi$,$\mu$);      ▷ Variable and value selection heuristic
combined
        dl = dl + 1;                                          ▷ : Increment dl for each
        $\mu = \mu \bigcup x$;                  ▷ Add literal x to the assignment trail $\mu$
        if (Boolean_Constraint_Propagation($\phi$, $\mu$) == CONFLICT) then
            $\beta$ = ConflictAnalysis($\phi$, $\mu$);      ▷ Analyze conflict and learn a clause
            if $\beta < 0$ then                                  ▷ $\beta$ is the backjump level
                return UNSAT;                                         ▷ Top-level conflict
            else
                backtrack($\phi$, $\mu$, $\beta$);                ▷ Backjump to start search again
                dl = $\beta$;
    return SAT;
```

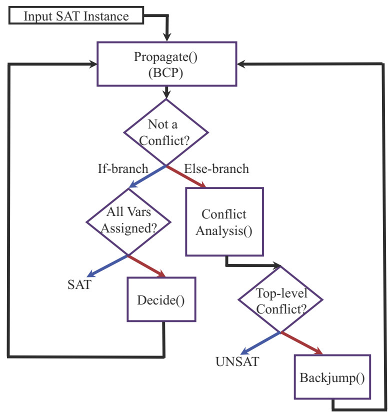
## 4. Conflict-driven Clause Learning SAT Solvers

In this Section we describe the conflict-driven clause-learning (CDCL) SAT solving algorithm[27,29,28] as given in Algorithm 2. Additionally, a diagrammatic presentation of the CDCL SAT solver algorithm is given in Figure 1 for improved readability. There are some minor differences between the presentations in Algorithm 2 and Figure 1, and hence we request the reader to go through both carefully. The Figure 1 also presents a search tree of assignments to an input formula. The white nodes in the tree represent decision variables, the dashed nodes are variables implied or propagated by BCP, and the dark leaf nodes represent conflict (unsatisfying assignments). The learned clauses capture the amount of search space pruned by the solver's learning scheme. For example, a learned clause with exactly one literal cuts down the search space by half, and with two literals cuts down the search space by 1/4 etc.

### *4.1.* Overview of CDCL Algorithm

We first describe the algorithm at a high level, followed by detailed descriptions of the key subroutines such as BCP, ConflictAnalysis, and variable and value selection (DecisionHeuristic) in the subsections that follow. Aside from a few minor differences, the version of the CDCL algorithm presented here is very similar to the one in the chapter on SAT solvers in the Handbook of Satisfiability. For ease of analysis, the CDCL algorithm presented here is in an iterative style as opposed to a recursive one.

The CDCL algorithm builds on top of the DPLL method, and differs from it primarily in its use of the following techniques: conflict analysis and clause learning, effective variable selection and value selection heuristics, restarts, clause deletion, and lazy



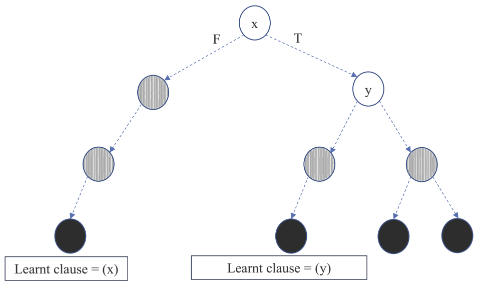


**Figure 1.** Left: The control-flow graph of a typical CDCL SAT Solver (presented in Algorithm 2). Right: A search tree of a unsatisfiable formula explored by a CDCL solver, where the nodes represent variables and edges correspond to value assignments (F for false, and T for true). The white nodes in the tree represent decision variables, the dashed nodes are variables implied or propagated by BCP, and the dark leaf nodes represent conflict (unsatisfying assignments). The missing sub-trees in the figure correspond to the space pruned by the learned clauses presented at the bottom of the tree.

data-structures. The CDCL algorithm takes as input a Boolean formula $\phi$ in CNF and an initially empty assignment $\mu$ (aka assignment trail), and outputs SAT if the input $\phi$ has a solution and UNSAT otherwise. Just like DPLL, the CDCL algorithm first calls the BCP subroutine on input formulas without having branched on any variables in it (line 5 in Algorithm 2). If a conflict is detected at this level (i.e., a top-level conflict) without having made any decisions, then CDCL returns UNSAT.

**Decision Levels and Assignment Trail:** On line 7 of Algorithm 2, the algorithm initializes the variable dl (abbrev. for *current decision level*) to 0. The variable dl keeps track of the number of decisions the solver makes as it traverses paths in the search tree of an input formula. Whenever a variable in the input formula is branched upon (a decision variable), dl in the CDCL algorithm is incremented by 1 (line 10). If the solver backjumps after ConflictAnalysis, the current decision level is also appropriately modified to the level that the solver backjumps to (line 18). The assignment trail (aka decision stack or partial assignment) $\mu$ is a stack data structure, where every entry corresponds to a variable, its value assignment, and its decision level. Whenever a variable $x$ is branched or decided upon, an entry corresponding to $x$ is pushed into the assignment trail. Whenever the solver backjumps from level $d$ to some level $d-\beta$, all entries with decision level higher than $d-\beta$ are popped from the assignment trail. The decision level of a variable is computed as follows: for unassigned variables it is initialized to -1. Unit variables in the input formula are assigned decision level 0. Whenever a variable is decided or branched upon, its decision level is set to dl + 1. Further, the decision level of an implied literal $x_i$ is the highest decision level of the implied literals in the unit clause $\omega$ that is the antecedent of $x_i$.

**Value Assignments and Antecedents:** In addition to the decision level, the solver maintains two other *dynamic* properties for every variable $x$, namely, the value $v \in \{0,1\}$ assigned to $x$ and its *antecedent*. All variables are initially assigned the value $u$. As the solver backjumps or restarts, the values these properties take may change. If a variable is a decision, then its value assignment is determined by the value selection heuristic in the DecisionHeuristic function. If, on the other hand the variable $x$ is implied, its value is determine by the unit clause (under the current partial assignment) that implies $x$. Such clauses are called antecedents. More precisely, the antecedent or the *reason clause* of a variable $x$ is the unit clause $c$ used by BCP to *imply* $x$. For variables that are decisions or unassigned, the antecedent is NIL.

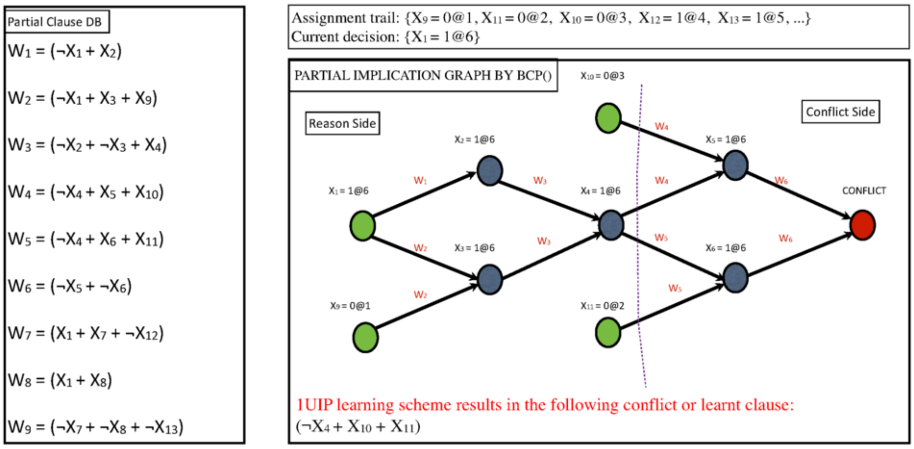
**The Search Loop in CDCL:** If there is no conflict at the top level, i.e., dl=0 (lines 5 to 7), then the algorithm checks whether all the variables of the input formula have been assigned a value (line 8). If so, the solver simply returns SAT. Else, it enters the body of the while loop on line 8, decides on a variable of the input formula using sophisticated variable and value selection heuristics (the DecisionHeuristic subroutine on line 9)[6], increments the decision level (line 10), pushes the decision to the partial assignment or the assignment trail $\mu$, along with its decision level (line 11), and performs BCP (line 12). If BCP returns CONFLICT (i.e., the current assignment $\mu$ is unsatisfying for the input formula), then conflict analysis is triggered. The ConflictAnalysis subroutine determines a reason or root cause for the conflict, learns a corresponding conflict or learned clause, and computes the *backjump level* (lines 13 to 18). The CDCL solver may jump back several decision levels in the search tree, unlike in the DPLL case where the solver backtracks only one level. If the backjump level is below 0, then the solver returns UNSAT since this corresponds to deriving false. For more on conflict analysis, see subsection 4.3.

## *4.2.* Implication Graph and BCP

Conceptually, the BCP procedure in CDCL is very similar to the one in the DPLL solver. We provide some additional details here regarding data structures such as the implication graph (see Figure 2 for an example) that are also relevant in the conflict analysis step of a CDCL solver. An implication graph $I$ is a directed acyclic graph maintained by the solver, whose nodes are variables of the input formula along with values and decision levels that they have been assigned by BCP under the current partial assignment. The leaves of this graph are decision variables and/or unit clauses in the input formula. The internal nodes are the implied variables. The edges are defined as follows: there is an edge from node $x_i$ to node $y$ if $x_i$ implies $y$. More precisely, given a node $y$ and its antecedent clause $w$, there is an edge from every variable $x_i$ in $w$, other than $y$, to $y$. The graph is updated by the solver as it makes decisions and derives unit clauses under the current partial assignment. We use the notation $x_i = v@d$ to denote that the value of $x_i$ is $v$ at decision level $d$.

The implication graph also has a special node $\perp$ that is used to represent the unsatisfied clause in the event of a conflict. The antecedent of this node $\perp$ is the falsified clause itself, and the edges to $\perp$ are constructed in the usual manner. It is easy to see that if literal l has an incoming edge labeled by clause c, then l must appear in c. Similarly, if literal l has an outgoing edge marked c, we can conclude $\neg l$ must appear in c.

[6] We assume that the function DecisionHeuristic performs both variable and value selection operations.



Partial Clause DB

$W_1 = (\neg X_1 + X_2)$

$W_2 = (\neg X_1 + X_3 + X_9)$

$W_3 = (\neg X_2 + \neg X_3 + X_4)$

$W_4 = (\neg X_4 + X_5 + X_{10})$

$W_5 = (\neg X_4 + X_6 + X_{11})$

$W_6 = (\neg X_5 + \neg X_6)$

$W_7 = (X_1 + X_7 + \neg X_{12})$

$W_8 = (X_1 + X_8)$

$W_9 = (\neg X_7 + \neg X_8 + \neg X_{13})$

Assignment trail: $\{X_9 = 0@1, X_{11} = 0@2, X_{10} = 0@3, X_{12} = 1@4, X_{13} = 1@5, \ldots\}$
Current decision: $\{X_1 = 1@6\}$

PARTIAL IMPLICATION GRAPH BY BCP()

1UIP learning scheme results in the following conflict or learnt clause:
$(\neg X_4 + X_{10} + X_{11})$

**Figure 2.** Left: A partial input clause database. Right: An implication graph for the clauses on the left. The nodes are denoted by the notation $x_i = v@d$, which denotes that the value of variable $x_i$ is $v$ at decision level $d$. The edges are marked by antecedent or reason clauses. Top: Assignment trail and the current decision.

## 4.3. Conflict Analysis in CDCL SAT solvers

The conflict analysis subroutine is unquestionably the most critical component of CDCL SAT solvers. It performs two key operations, namely, clause learning and determining the level to which the solver needs to backjump to after a conflict has been analyzed. During its execution, when a CDCL solver determines that the current partial assignment to an input formula is unsatisfying (i.e., is a *conflicting or no-good assignment* that falsifies at least one clause in the input formula), the ConflictAnalysis subroutine is invoked to examine why this is so. The root cause analysis performed by ConflictAnalysis returns one or more clauses, referred to as learned clauses, that are stored alongside the input formula [7]. These learned clauses (aka conflict clauses) have the property that they prevent the solver from subsequently exploring not only the conflicting assignment, but also potentially exponentially many similar assignments, thus cutting down the search space dramatically. This is sometimes referred to as search space pruning. Further, the learned clauses also have the property that they are implied by the input formula. In addition to computing the learned clause, the conflict analysis procedure also computes the decision level to which the solver backjumps to, thus undoing the decisions that led to the conflict.

### 4.3.1. Clause Learning

Perhaps one of the simplest clause learning method developed to-date is the Decision Learning Scheme (DLS). In this method, the learned clause is defined as the disjunction of the negated decisions that led to a conflict. Consider a formula $\phi$ over variables $x_1, x_2, \ldots, x_n$, and assume that the solver has made the following decisions $x_1, \neg x_2, \neg x_3$.

[7]For ease of management, most modern solvers store learned clauses in a database separate from the clauses of the input formula. Having said that, conceptually we can assume that all clauses are stored together.

Further, assume that only the decisions $x_1$ and $\neg x_2$ led to the conflict. The decision $\neg x_3$ didn't participate in the conflict, i.e., does not appear in the implication graph. Mathematically, we can write this as $\phi \wedge x_1 \wedge \neg x_2 \vdash \perp$, or put differently, $\phi \vdash (\neg x_1 \vee x_2)$. The DLS then outputs the clause $(\neg x_1 \vee x_2)$ as the learned clause. While DLS is very simple to understand and implement, the clauses thus produced have been empirically observed to be of *poor quality*, i.e., they don't prune the search space as effectively as other learning schemes like 1UIP that we discuss below. To better understand the more sophisticated clause learning methods, we present two distinct but complimentary ways of describing them, namely, as cuts in the implication graph and as a resolution proof system.

**Clause Learning as Cuts in Implication Graphs:** Clause learning procedures can very naturally be viewed as methods that construct learned clauses by computing cuts on the implication graph such that all decision variables are on one side of the cut (the reason side) and the conflict node is on the other (the conflict side), and the learned clause is constructed by taking the disjunction of the negation of literals immediately to the left of the cut. In this view, these methods yield conflict clauses of differing quality depending on where the implication graph is cut.

**1UIP Clause Learning Scheme via Cuts in Implication Graphs:** Perhaps the most effective and important clause learning method is the First Unique Implicant Point (1UIP) scheme[29]. The term 1UIP refers to a node $N$ in the implication graph where all paths from the current decision node (the node corresponding to the decision variable at the highest decision level at the time of the conflict) to the conflict node must go through $N$ and it is the closest such node to the conflict node. $N$ is a dominator node for all paths from the current decision node to the conflict node. In the 1UIP clause learning scheme, the implication graph is cut immediately to the right of the 1UIP node, such that this node and all the decision variables are on the left side of the cut and the conflict node is to the right side. The 1UIP clause is constructed by taking the disjunction of the negation of literals immediately to the left of the cut.

**1UIP Clause Learning via the Resolution Proof System:** While the above view is a useful way of understanding clause learning schemes through the lens of implication graph and cuts, the method by which the ConflictAnalysis subroutines actually produce learned clauses is best understood in terms of the general resolution proof system. The only rule in the general resolution proof system (or simply, resolution) is the following:

$$(\beta \vee x) \quad (\neg x \vee \alpha) \vdash (\beta \vee \alpha)$$

where $\beta$ and $\alpha$ are disjunctions of literals and may not have any occurrence of $x$.

Although this view of clause learning as a sequence of proof steps applies to any method, we will specifically describe the 1UIP clause learning scheme in this way. The sequence of resolution proof steps, that start at the conflict node of the implication graph and terminate with the computation of the 1UIP conflict clause.

The 1UIP clause-learning procedure starts by setting the conflicting clause (i.e., the falsified clause that labels the incoming edges to the conflict node) as the "current" conflict clause. It then performs resolution over all variables assigned at the current decision level (the one at which the conflict occurred), thus deriving intermediate conflict clauses after each resolution step, until the 1UIP node is the only one from the current decision level. Each resolution step is performed as follows: for any "current" clause B, the procedure first computes the last assigned literal[8] $l$ in $B$ and chooses some incoming edge to $l$ labeled with $B'$. It then resolves $B$ and $B'$, and the resolvent thus obtained is marked as the current clause. This procedure is repeated for all "$l$" assigned at the highest decision level at which the conflict occurred, until the current clause contains exactly one literal from the current decision level, namely, the negation of the 1UIP node.

In the context of our example in Figure 2, the 1UIP clause learning procedure starts by setting the "current" conflict clause to be $W_6$. It then chooses the last assigned literal, say, $x_5$ at decision level 6, and resolves its antecedent $W_4$ and $W_6$. The resultant clause $\neg x_4 \vee \neg x_6 \vee x_{10}$ is now set as the current conflict clause. The procedure then moves to the next to last literal set at the current decision level, $x_6$. It resolves the current conflict clause $\neg x_4 \vee \neg x_6 \vee x_{10}$ and $W_5$ to obtain $\neg x_4 \vee x_{10} \vee x_{11}$. Having resolved over all variables at the current decision level in the current conflict clause except the 1UIP node, and having the 1UIP node in the current clause, causes the procedure to terminate and output the current conflict clause. Besides 1UIP, researchers have explored other kinds of clause learning schemes such as learning all UIP clauses (e.g., the GRASP SAT solver[27]). However, 1UIP clause learning has proven to be the most effective for many classes of industrial applications.

**Clause Learning and Backjumping:** In Section 3 on DPLL SAT solvers we described the simplest form of backtracking, wherein, upon reaching a conflict, these solvers undo the last decision that led to the conflict, which may lead the solver to backtrack to the previous decision level and continue its search. In the context of CDCL solvers many backtracking methods have been explored. Perhaps the most well-known is called non-chronological backtracking (or simply, backjumping), wherein, the solver backjumps to the second highest decision level over all the literals in the 1UIP learned clause. This has the added benefit that the 1UIP is now unit clause under the "current" partial assignment post backjump. Such learning schemes are also called *asserting*.

## *4.4.* Machine Learning and Variable Selection Heuristics in SAT solvers

Along with clause learning schemes, variable and value selection heuristics are perhaps the most important components of efficient SAT solvers[21]. While there has been considerable research on variable selection heuristics over the years, the VSIDS (Variable State Independent Decaying Sum) method[29] is one of the dominant ones. Over the last decade, it has become almost customary to start any description of variable selection heuristics by describing the VSIDS method. This is not surprising since for nearly 20 years the VSIDS method has been the dominant variable selection heuristic and is one of the key reasons behind the success of SAT solvers. However, we shall deviate from this practice in order to illustrate a much deeper lesson as it pertains not only to the design of variable selection heuristics, but also the design of solvers in general. Specifically, we shall first discuss the machine learning-based LRB variable selection heuristic introduced by Liang and G.[24], that has had profound impact on both the design of variable selection heuristics, as well as many other heuristics such restarts and clause deletion policies.

[8]Recall that BCP enforces a total order on assigned literals

### *4.4.1.* The Setting: Understanding CDCL Solvers via Machine Learning

The question of why SAT solvers are efficient for a wide variety of applications has been one of the most important questions in solver research. After all, given that the SAT problem is NP-complete, we don't expect any algorithm to scale to even hundreds of variables, let alone millions of variables and clauses for formulas obtained from real-world applications. How is this possible? As we hinted earlier in the Introduction, answering this question would require us to look at solvers through the lens of both proof systems and machine learning.

While we don't have a complete picture of why SAT solvers are so efficient, we have enough clues to suggest that the "solver as a proof system" is a powerful view that affords us deep insights into the working of solvers and enables us to better navigate the design space of solver algorithms. More precisely, we get two advantages from this view. First, we can use proof complexity to prove strong lower and upper bounds on solvers. We will expand on this further in Subsection 4.6. Second, we can view a solver as an implementation of a proof search algorithm for a proof system. Put differently, an implementation of a solver would consist of proof rules and methods for optimal sequencing and selection of these rules for various classes of inputs. Given that current solvers produce copious amounts of data as they search for proofs and/or solutions, it further suggests that such optimization procedures may best be designed by leveraging machine learning algorithms.

When viewed from this perspective, the CDCL SAT solver can be seen as having the following structure: the clause learning subroutine corresponds to the implementation of the general resolution proof rule (a teacher), and the BCP and the variable and value selection heuristics can be seen as an actor (a student) that interacts with the clause learning component in a corrective feedback loop with the aim of optimizing some reward (e.g., minimize solver runtime). This student-teacher model with corrective feedback loop is very reminiscent of how reinforcement learning (RL) methods work, where there is an actor, who continually interacts with her environment, attempting to optimize a reward. In fact, what we show below is that reinforcement learning methods are a powerful way to design many solver heuristics, and in particular this view of branching methods as actors in a RL setting led to the development of the Learning Rate Based (LRB) variable selection heuristic.

### *4.4.2.* The LRB Variable Selection Heuristic

Any variable selection heuristic can be viewed as solving the optimization problem of ranking variables of the input formula such that the overall solver runtime is minimized. Additionally, most modern variable selection heuristics are dynamic, that is, they are called at regular intervals throughout the run of the solver, and may provide distinct rankings at different invocations. The solver typically picks the top-ranked variable in this ranking, and branches on it. An additional requirement placed on the design of variable selection heuristics is that they must be cheap to compute.

Unfortunately, the task of designing variable selection heuristics that, given any input, minimize solver runtime by picking just the right variables and have very little computational overhead is simply too hard to solve. Instead, solver designers often use metrics that are proxies for minimizing solver runtime, i.e., ones that empirically correlate well with minimizing solver runtime and are cheap to compute. Put differently, the problem of designing a solver heuristic reduces to identifying the appropriate optimization problem, along with the corresponding proxy metric, and coming up with methods to solve said problem.

In the case of the Learning Rate Branching (LRB) heuristic[24,22], the authors identified the optimization problem as maximizing the Global Learning Rate (GLR) measure (a proxy for minimizing solver runtime), and the method to solve this optimization problem as Exponential Recency Weighted Average (ERWA) method from the RL literature [9].

**Global Learning Rate (GLR):** The GLR metric is simply a ratio of the number of conflict clauses learned and the number of decisions made by the solver, at any point during the run of the solver. The GLR is a good metric for the variable selection optimization problem since it is very cheap to compute and has been empirically shown to correlate strongly with minimizing solver runtime. The intuition behind the choice of GLR is that solvers want to maximize the number of learned clauses (which corresponds to *learning about* the search space of the input formula) while simultaneously minimizing the number of decisions (or assumptions used to derive these clauses). Ideally, one would also like to maximize some kind of quality metric with regard to learned clauses, in addition to GLR. Fortunately, existing 1UIP clause learning scheme already seems to produce "high quality" clauses, at least based on empirical observations.

**The Multi-Arm Bandit Problem:** The Multi-Arm bandit (MAB) is a classic RL problem, which is best described in terms of a gambler playing $n$ slot-machines (the multi-arm bandits) in a casino. The objective of the gambler or actor is to maximize her reward or monetary earnings, with minimal expenditure of resources. Each slot machine has a *reward probability distribution* which associates a probability with every possible value of reward. Obviously, the actor is not privy to this distribution. In a more complex version of this problem, more relevant in our setting, this probability distribution may change with time. Such distributions are called non-stationary.

At any point in time the actor has *$n$ actions* to choose from, corresponding to playing one of the $n$ slot machines. The actor picks an action, plays the corresponding machine and receives a monetary reward. The objective for the actor is to find the sequence of actions that maximize her global reward and minimize cost due to resource usage. The actor may randomly play few slot machines to see which one maximizes her reward (exploration), and then focus on the one that has the largest payout, at least, for as long it lasts (exploitation). If the slot machine's payout starts decreasing, the actor may move to other slot machines. This game of exploration vs. exploitation continues until some kind of timeout or other termination condition is reached. In RL literature, this approach to solving the non-stationary MAB problem is codified as the Exponential Recency Weighted Average (ERWA) method, which we discuss below in detail. Interestingly, a very similar approach also works well for variable selection in CDCL solvers.

[9]It goes without saying that these approximations of ideal branching heuristics are bound to perform well on some class of instances and poorly on others. There are no guarantees of optimality here. The point simply is that RL methods seem to work really well in practice when applied in the SAT solver setting, especially given the fact that solvers produce so much data about the search space they explore. Further, this correspondence between solver heuristics and machine learning methods offers solver designers a new way of navigating the design space of solver algorithms by leveraging the rich literature on machine learning.

**MAB and LRB Branching Heuristic:** As alluded to above, the correspondence between CDCL and RL (specifically, non-stationary MABs) is very strong indeed: variable selection heuristics correspond to actors, the act of variable selection or branching corresponds to actions in RL parlance, the variables of the input formulas correspond to the slot machines, clause learning corresponds to the environment which provides corrective feedback and the learned clauses correspond to *local* rewards, and the optimization metric GLR corresponds to reward. The actor (LRB) chooses variables that maximize its reward (GLR). While maximizing the GLR remains the global objective of the actor (branching heuristic), every action also has an associated *local reward*, referred to simply as *learning rate*, which measures the contribution to GLR by every individual action (branching on a variable). Below we describe the concept of learning rate that the SAT solver keeps track of for every variable in the input formula.

**Learning Rate:** In order to better understand the learning rate we first recall how clause learning works: conflict clauses are computed during conflict analysis via a series of resolution steps performed on the clauses that mark edges of the implication graph under analysis. We say that a variable $v$ participates in generating a learned clause $l$ if either $v$ appears in $l$ or $v$ appears on the conflict side of the implication graph under analysis. In other words, $v$ plays an essential role in learning the clause $l$ from the the encountered conflict. We define a variable $I$ as the interval of time between the point $v$ is assigned to the point it may become unassigned due to backjumping. Let the predicate $P(v,I)$ be the number of learned clauses in which $v$ participates during the interval $I$ and let $L(I)$ be the number of learned clauses generated in the interval $I$ (which may or may not contain $v$). The Learning Rate (LR) of variable $v$ during the interval $I$ is defined as $\frac{P(v,I)}{L(I)}$, i.e., it computes the ratio how many learned clauses $v$ participated in during the interval $I$ and how many total clauses were produced during the same interval. In other words, the learning rate of a variable measures how frequently the variable has triggered conflict when assigned during a solver run. It is clear that branching on variables with high learning rate result in higher GLR than otherwise.

**Learning Rate Branching:** We are now ready to explain the LRB heuristic, through the lens of ERWA heuristic for MAB problems.

Recall that the branching heuristic (the actor) chooses one variable to branch on from a set of $n$ variables (the action). In order to maximize the GLR, the actor maintains the history of rewards (learning rate) it received from each variable. This is maintained as a time series. For every time step (suitably defined), for every variable, its learning rate is recorded for that time step. The actor then computes an Exponential Moving Average (EMA) over this time series for all variables, and chooses the variable with the highest EMA over the learning rate time series.

The idea of an EMA is simple yet very powerful, and is used in many fields to discover "recent" trends in time series data (e.g., in finance for stock picking), and in RL (e.g., to solve the MAB problem). Consider a scenario where you have $n$ different time series (e.g., $n$ variables with their learning rates over time, one time series per variable. Similarly, payouts from $n$ slot machines over time). The problem for the branching heuristic (gambler in MAB) in this scenario is which variable to pick next (which slot machine to play next) to maximize its reward (money payouts). Further, in this setting the learning rates are non-stationary, i.e., the reward probability distribution changes over time.

One approach to solve this problem could be to focus on the variable that has had the highest learning rate since the beginning of the solver's run (similarly, always pick the slot machine at time *t* with the largest total payout up until *t*). Such an approach requires the branching heuristic (actor) to maintain an average since the beginning of the time series for each variable (slot machine). The trouble with this approach in a non-stationary setting is that it doesn't account for potential fall in learning rate (resp. payouts) in "recent" time intervals. A better approach would be to focus on those variables whose learning rate is trending up in "recent" time intervals, as opposed to ones whose total average is highest since the beginning of the time series. This kind of average, that gives more weight to recent data points in a time series than older ones, is called Exponential Moving Average (EMA). It is called an exponential moving average since the window of time over which the average is computed changes in a manner where older data points in the time series are exponentially decayed away as a function of time, with those data points that further away from the "current" time getting decayed away more aggressively.

As mentioned above, the solver maintains a time series of learning rate for every variable throughout its entire run. It then computes an EMA on these time series data over all variables, decaying older data points exponentially relative to more recent data points. The variables are then ranked based on the EMA of their learning rates, and the highest unassigned variable in this ranking is branched upon when the LRB heuristic is invoked. As the time series data continues to accumulate the ranking may change dynamically.

**VSIDS and LRB:** Another view of LRB (that also applies to VSIDS) is that both LRB and VSIDS maintain a value called *activity* for every variable of an input formula. The activity of all variables is initially set to 0. In VSIDS, the activity of all variables that appear on the conflict side of the implication graph is bumped by some constant (which corresponds to reward in RL), and these activities are decayed by a constant between 0 and 1 at regular intervals. It is this decaying by a constant that essentially corresponds to an EMA[25].

By contrast, in LRB, whenever a variable transitions from assigned to unassigned the LR is calculated (this corresponds to reward) and is appended to the time series. When the solver requests the next branching variable, LRB computes the EMA over the time series of all variables and picks the one with the highest value (the activity). LRB has undergone extensive testing over application, hard-combinatorial, and cryptographic instances from many years of SAT competition benchmarks. It has proven itself to be more effective than VSIDS for many classes of instances and displaced VSIDS as the best known branching heuristic in nearly 20 years.

## *4.5.* Restarts in CDCL Solvers

The idea of solver restarts[18] is quite straightforward: at regular intervals during its run, a solver may discard the search tree it is exploring. This might seem counter-intuitive at first glance, but restarts have proven to be very effective in helping improve solver performance in practice. When the solver restarts, it doesn't throw away all its state. Solvers typically preserve all the clauses they have learned up until the restart is triggered (assuming no separate clause deletion policy is in place), as well as the activity of all variables. The only part of the solver state that is deleted is the assignment trail, causing the solver to throw away the search tree it has built since the previous restart.

Many explanations have been proposed to explain why restarts are so useful. We still don't have a clear picture. However, recent empirical work has shown that some of the existing hypotheses are clearly false. For example, the "heavy-tailed explanation" was proposed as a possible reason why restarts are so effective[18]. The idea has its root in Las Vegas algorithms, whose running times are typically characterized by a probability distribution. It is possible that depending on the random seed used such algorithms may terminate quickly on some instances. The idea is that these algorithm, upon a restart, "randomly jump" to an easy part of the search space.

However, this explanation of restarts doesn't quite apply to CDCL SAT solvers. For one, solvers are deterministic algorithms unlike Las Vegas algorithms. Further, the definition of restarts in the context of Las Vegas algorithms differs significantly from that of restarts. In the case of Las Vegas algorithms, restarts erase the state of the algorithm (suitably defined) and start the search from scratch. This is not how restarts work in the context of CDCL SAT solvers, since significant aspects of the state are preserved across restart boundaries.

A better explanation, which has empirical support in the work on Liang et al.[26], is that restarts "compact" the assignment trail resulting in higher-quality conflict clauses. The crucial intuition is that when a solver restarts frequently, it has to rebuild the search tree often and this leads to conflict clauses appearing at lower depths in the assignment trail. Such clauses that are learned at "compacted" assignment trails tend to be shorter, and perhaps more importantly, have smaller LBD (Literal Block Distance). The LBD is a metric that measures the number of decision level a conflict clause spans at the time during the run of the solver that it is learned. That is, the variables in the clause are bucketized by their decision levels and the number of such buckets is the LBD. It has been empirically observed that clauses with lower LBD tend to be of "higher quality", i.e., result in lower solver runtime. Liang et al. go further, and propose a machine learning based restart policy that uses LBD as the optimization metric. The result is one of the most competitive restarts policy to-date, and a deeper understanding of why restarts are effective.

**Clause Deletion, Phase Saving, and Other Hueristics:** So far we have discussed in clause learning, variable selection, and restart heuristics in great detail. Solvers employ a myriad of other heuristics such as clause deletion, phase saving, inprocessing etc. While these techniques are empirically important, they are not as universally critical as clause learning, variable selection, and restarts. Further, they are conceptually quite easy to understand. Hence, we refer the reader to the Handbook of Satisfiability as an appropriate reference for more details on these other heuristics.

## *4.6.* Proof Complexity of SAT Solvers

In this subsection we show that CDCL and general resolution simulate each other, provided that the CDCL solver is given a (non-deterministic) sequence of variable choices in advance and it has a (i.e. *asserting*) clause learning scheme. The ideas presented here are from the seminal paper by Pipatsrisawat and Darwiche [34]. We use an extended proof to show that $\mathsf{CDCL}(T)$ simulates the Res(T) proof system. In fact, in this section we provide a high level intuition of the proof, and follow up with the more detail when we describe the extended proofs in $\mathsf{CDCL}(T)$ context in Section 5.2.

Recall that an *assignment trail* is a sequence of pairs $\sigma = \{(\ell_1, d_1), (\ell_2, d_2), \ldots, (\ell_t, d_t)\}$ where each literal $\ell_i$ is a literal from the theory and each $d_i \in \{\mathsf{d}, \mathsf{p}\}$, indicating that the literal was set by the solver by a decision or a unit propagation respectively. The *decision level* of a literal $\ell_i$ in the branching sequence is the number of decision literals occurring in $\sigma$ up to and including $\ell_i$. The *state* of a CDCL solver can be defined as $(\mathscr{F}, \Gamma, \sigma)$, where $\mathscr{F}$ is the input CNF formula, $\Gamma$ is a set of learned clauses, and $\sigma$ is an assignment trail. Given an assignment trail $\sigma$ and a clause $C$ we say that $C$ is *asserting* if it contains exactly one literal occurring in $\sigma$ of decision level $|C|$. A clause learning scheme is *asserting* if all conflict clauses produced by the scheme are asserting with respect to the assignment trail at the time of conflict.

A *extended branching sequence* is an ordered sequence $B = \{\beta_1, \beta_2, \ldots, \beta_t\}$ where each $\beta_i$ is either (1) a branching literal, or (2) a symbol $x \in \{\mathsf{R}, \mathsf{NR}\}$, to denote a restart or no-restart, respectively. If $A$ is a CDCL solver, we use an extended branching sequence to dictate the operation of the solver $A$ on $\mathscr{F}$: whenever the solver calls the Branching Scheme, we consume the next $\beta_i$ from the sequence. If it is a literal, then we branch on that literal appropriately, otherwise restart as dictated by the extended branching sequence. If the branching sequence is empty, then simply proceed using the heuristics defined by the algorithm.

We now introduce a central notion that was originally defined by Pipatsrisawat and Darwiche [34] and separately Atserias, Fichte and Thurley [2].

**Definition 4.1** (Empowering Clauses). Let $\mathscr{F}$ be a set of clauses and let $A$ be a CDCL solver. Let $C = (\alpha \Rightarrow \ell)$ be any clause. We say that $C$ is *empowering with respect to $\mathscr{F}$ at $\ell$* if the following holds: (1) $\mathscr{F} \models C$, (2) $\mathscr{F} \wedge \alpha$ is unit consistent, and (3) any execution of $A$ on $\mathscr{F}$ that falsifies all literals in $\alpha$ does not unit propagate $\ell$. The literal $\ell$ is said to be *empowering*. If item (1), (2) are satisfied but (3) is false then we say that the solver $A$ and $\mathscr{F}$ *absorbs* $C$ at $\ell$; if $A$ and $\mathscr{F}$ absorbs $C$ at at every literal then the clause is simply *absorbed*. (The concepts of empowering and absorbed are duals of each other.)

One should think of the absorbed clauses as being "learned implicitly" — absorbed clauses may not necessarily appear in $\mathscr{F}$. However, if we assign all but one of the literals in the clause to false then unit propagation in CDCL($T$) will set the final literal to true. That is, even if the absorbed clause $C$ is not in $\mathscr{F}$, the unit propagation sub-routine behaves "as though" the absorbed clause is actually in $\mathscr{F}$.

Symmetrically, in order for a clause $C$ to be learned by a DPLL solver, it must be *empowering at some literal $\ell$* at the time it is learned. To see this, consider a trace of a DPLL solver wherein we have just learned a clause $C$. Since we have learned $C$ it easy to see that it must be the case that $\mathscr{F} \models^T C$. Let $\sigma$ be the branching sequence leading to the conflict in which we learned $C$, and let $\ell$ be the last decision literal assigned in $\sigma$ *before* the solver hit a conflict (if DPLL uses an asserting clause learning scheme, such a literal must exist). We can write $C \equiv (\alpha \Rightarrow \neg\ell)$, and clearly $\alpha \subseteq \sigma$. Thus, at the point in the branching sequence $\sigma$ before we assign $\ell$ it must be that $\mathscr{F} \wedge \alpha$ is unit consistent, since we have assigned another literal after assigning each of the literals in $\alpha$. Finally, $\mathscr{F} \wedge \alpha \nvdash_1^T \ell$ since $\neg\ell$ was chosen as a decision literal *after* we set the literals in $\alpha$.

**Definition 4.2** (1-provable Clauses). Given a CNF $\mathscr{F}$, clause C is 1-provable with respect to $\mathscr{F} \iff \mathscr{F} \wedge \neg C \vdash_1 \mathit{false}$.

Put differently, a clause $C$ is 1-provable with respect to a CNF $\mathscr{F}$, if $C$ is derivable from $\mathscr{F}$ only using BCP.

**Theorem 4.1.** *CDCL polynomially simulates general resolution.*

The high level idea of the simulation is as follows: We need to show that for any general resolution proof for an unsatisfiable formula, the CDCL solver can "simulate" that proof with only polynomial overhead in the proof size (in terms of number of clauses). The crucial insight here is that for formulas $\mathscr{F}$ for which BCP alone cannot establish unsatisfiability, there exist empowering clauses implied by $F$ which when added to $\mathscr{F}$ cause BCP to correctly determine that this formula is UNSAT. Further, given a general resolution proof $P$ of $\mathscr{F}$, it contains at least one empowering and 1-provable clause in it. It further turns out that there exists an non-deterministic extended branching sequence for CDCL solvers with asserting learning schemes such that they can absorb such a clause. This process is repeated until there are no more clauses that need to be absorbed, and we have shown that CDCL simulates general resolution[10]. (The reverse direction is immediate.)

## 5. Lazy SMT Solvers via CDCL(T)

In this Section, we provide an overview of the architecture of SMT solvers, followed by proof complexity theoretic analysis of their power as proof systems. We will describe SMT solvers along our now familiar line of "solvers as proof systems". We deliberately deviate from traditional surveys of SMT solvers, where a high-level overview of SMT solver algorithm is followed by details about individual theories. Our description provide a complementary description to existing surveys.

SMT solvers are powerful algorithms designed to solve the satisfiability problem for first-order theories that are relevant in the context of program analysis, testing, security, and verification. They support a much richer input language than SAT solvers, e.g., first-order theories such as non-linear arithmetic over the reals and integers, theories over strings and regular expressions, bit-vectors, arrays, uninterpreted functions, datatypes, and floating-point arithmetic. The impact of SMT solvers on many areas of software engineering and security has been dramatic and well-documented.

Generally speaking, there are two categories of SMT solvers, *eager* and *lazy*. Eager solvers are based on the idea of equisatisfiable reductions of input formulas in one theory into another (e.g., Bit-vectors to Boolean logic), and then invoking the appropriate solver on the resultant formula. The appeal of such an approach is that it leverages advances in solvers for one theory (e.g., SAT solvers) to efficiently solve formulas in another (e.g., bit-vectors). However, such an approach may introduce costs in terms of blowup in formula size during translation from one theory to another.

By contrast to eager solvers, lazy SMT solvers are based on two key ideas, namely, abstraction-refinement and specialized decision procedures for first-order theories such as uninterpreted functions, arithmetic, strings, bit-vectors etc. At its core, a lazy SMT solver is a SAT solver extended with decision procedures for these first-order theories

[10]In the Subsection 5.2 we give a formal and more general version of this proof for $\mathsf{CDCL}(T)$, that also applies to the CDCL SAT solver case.

**Algorithm 3** The CDCL(T) Architecture of SMT Solvers

```
1:  input: CNF formula $\mathscr{F}$ over $T$-literals
2:  SAT or UNSAT
3:  Let $\sigma = \emptyset$ be an initially empty partial assignment of $T$-literals
4:  $\Gamma$ be an initially empty collection of learned clauses
5:  while true do
6:      Perform BCP on Boolean abstraction, until saturation, and update $\sigma$
7:      Apply the $T$-propagate scheme, and update $\sigma$ accordingly
8:      if (Boolean abstraction of $\mathscr{F}$ and $\Gamma$ and $\sigma$ together result in a conflict) then
9:          if ($\sigma = \emptyset$) then
10:             return UNSAT
11:         Apply the clause learning scheme to learn a conflict clause $C$, add it to $\Gamma$
12:         Backjump $\sigma$ to the second highest decision level in $C$
13:     else if ($\sigma \vdash^T \emptyset$) then
14:         Apply the $T$-conflict scheme to learn a conflict clause $C$, add it to $\Gamma$
15:         Backjump $\sigma$ to the second highest decision level in $C$
16:     else
17:         if $\sigma$ satisfies $\mathscr{F}$ then
18:             return SAT
19:         Apply the restart scheme to decide whether or not to restart
20:         if restart then
21:             Set $\sigma = \emptyset$
22:             Restart loop
23:         Apply the branching scheme to choose a decision literal $\ell$, $\sigma = \sigma \cup \{\ell\}$
```

that enhance the SAT solver's propagation and conflict analysis subroutines (they are also sometimes referred to as CDCL(T) solvers, as described below). Informally, given a formula $\phi$ over first-order theories, the SMT solver first constructs a Boolean abstraction (an over-approximation) $\phi_b$ of $\phi$. If the abstraction is UNSAT, the solver returns UNSAT. Else, there is some satisfying assignment $A_b$ for $\phi_b$. The SMT solver then essentially verifies this assignment by constructing a conjunction of theory literals corresponding to literals in $A_b$, and calling the appropriate theory decision procedure on it. This abstraction-refinement process repeats until the solver correctly determines the satisfiability of the input formula. (We describe this in greater detail below.)

## *5.1.* Understanding the Architecture of CDCL(T)

A one line description of SMT solvers is that they are "CDCL SAT solvers extended with theory solvers" as shown in Algorithm 3, and hence they are often referred to as DPLL(T) solvers. The idea of DPLL(T) was first introduced in a landmark paper by Ganzinger, Hagen, Nieuwenhuis, Oliveras, and Tinelli[17], and then refined further in a subsequent paper by Nieuwenhuis, Oliveras, and Tinelli[31]. The DPLL(T) architecture is sometimes also referred to as CDCL(T). We believe that the term CDCL(T) is a better fit. The reason is that prior to mid-2000's, the term DPLL was used to describe two very different kinds of SAT solvers, namely, one that was invented by Davis, Putnam, Loveland, and Loeggemann in 1960's (and hence the name DPLL), as well as the one invented in the late 90's which enhanced DPLL with clause-learning (what we now call as CDCL). However, from a proof complexity point of view, the difference between DPLL and CDCL is vast. DPLL is polynomially equivalent to tree-like resolution, while CDCL is polynomially equivalent to the much stronger general resolution. By the same token, we believe the term CDCL(T) better captures modern SMT solvers than the term DPLL(T). Perhaps the most effective way to understand CDCL(T) is that this architecture[11] extends the CDCL algorithm with a solver T for combination of various first-order theories, wherein, this theory solver T is used to enhance the conflict analysis and propagation of the CDCL algorithm.

CDCL(T) solvers take as input CNF formula $\mathscr{F}$ over theory literals (written as $T$-literals), and returns SAT or UNSAT. The CDCL(T) initially performs reasoning over a Boolean abstraction of the formula $\mathscr{F}$, i.e., every theory literal is abstracted appropriately by a Boolean literal and the resultant Boolean formula is analyzed by the SAT solver inside the CDCL(T) solver (lines 6 to 8). If this formula is UNSAT and the partial assignment $\sigma$ is empty, clearly there is no reason to further analyze $\mathscr{F}$ (lines 9 and 10), and the SMT solver returns UNSAT. On the other hand, if $\sigma$ is non-empty the solver learns a conflict clause over the variables in the Boolean abstraction (lines 11 to 12).

Else, the Boolean abstraction is SAT. This does not imply that the input formula is satisfiable, for it is possible that the abstraction has a satisfying assignment but this is not satisfying when the theory is taken into account. Hence, the solver checks whether this partial assignment is consistent with the theory solver. On lines 13 to 15, the solver calls the appropriate theory solver to check whether the partial assignment $\sigma$ is consistent with the theory. If not, it invokes the theory conflict analysis subroutine $T$-conflict to identify theory conflicts and add it to $\gamma$.

On lines 17 to 18, the solver checks if the partial assignment is $T$-satisfying. If so, return SAT. Elsif, depending on the restart scheme, the solver may choose to restart (lines 20 to 22). If the control has reached line 23, it means the solver has found no conflicts (either Boolean or theory-level), has not yet found a complete satisfying assignment, and BCP and $T$-propagate have saturated. This causes the solver to call the variable and value selection heuristics, and branch on a literal.

## *5.2.* **CDCL(T) and** $\mathsf{Res}(T)$

In this Section we show that lazy SMT solvers and resolution modulo theories are polynomially-equivalent as proof systems, provided that the SMT solvers are given a set of branching and restart decisions *a priori* based on the work of Robere et al.[35]. We model SMT solvers by the algorithm schema[12] $\mathsf{CDCL}(T)$, given in Algorithm 3. Using this schema we prove two results: first, if the theory solver in $\mathsf{CDCL}(T)$ can only reason about literals occurring in its input formula, then $\mathsf{CDCL}(T)$ is polynomially equivalent to the proof system $\mathsf{Res}(T)$. Second, if the theory solver is strengthened so that it is allowed to introduce new literals then the resulting solver can polynomially simulate $\mathsf{Res}^*(T)$. The proofs of these results use techniques developed for establishing polynomial equiv-

[11]We use the term architecture to highlight the fact that CDCL(T) is a set of algorithms, depending on the theory T.

[12]In the literature, SMT solvers are typically defined as abstract state-transition systems (see, for instance, [17,6]); we have chosen to define it instead as an algorithm schema (cf. Algorithm 3) inspired by the abstract definition of a $\mathsf{CDCL}$ solver by Pipatsrisawat and Darwiche [34].

alence between Boolean CDCL solvers and resolution by Pipatsrisawat and Darwiche [34].

If $T$ is a theory and $A, B$ are formulas over $T$ then we write $A \vdash^T B$ as a shorthand for $T \cup \{A\} \vdash B$ (i.e., every model of the theory $T$ that satisfies $A$ also satisfies $B$). We also define *unit resolution* for theories, which describes the action of the *unit propagator*.

As alluded to above, the $\mathsf{CDCL}(T)$ system is in fact a collection of algorithm, each defined by specifying algorithms for each of the bolded "schemes" in Algorithm 3. The definitions of clause learning scheme, restart, and branching schemes are standard from CDCL SAT literature (See Section 4 above). For $T$-propagate and $T$-conflict, we provide brief definitions below:

**$T$-Propagate Scheme:** During search, the $\mathsf{CDCL}(T)$ solver can hand the theory solver the current partial assignment $\sigma$ and ask whether or not it should unit-propagate a literal; if a unit propagation is possible the theory solver will return a clause $C$ from the theory witnessing this unit propagation.

**$T$-Conflict Scheme:** When the theory solver detects that the current partial assignment $\sigma$ contradicts the theory, the **$T$-Conflict Scheme** is applied to learn a new clause of literals $C$, $\neg C \subseteq \sigma$, which is added to the clause database.

We pay particular interest to the specification of the $T$-propagate scheme. The next definition describes two types of propagation schemes: a *weak* propagation scheme is only allowed to return clauses which propagate literals in the formula, while the more powerful *strong* propagation scheme returns a clause of literals from the theory that may contain new literals.

**Definition 5.1.** A *weak $T$-propagate scheme* is an algorithm which takes as input a conjunction of theory literals $\sigma$ over $T$ and returns (if possible) a clause $C = \neg\sigma \vee \ell$ where $T \models C$ and the literal $\ell$ occurs in the input formula of the $\mathsf{CDCL}(T)$ algorithm.

A *strong $T$-propagate scheme* is an algorithm which takes as input a conjunction of literals $\sigma$ over $T$, and if possible returns a clause $C$ of literals from $T$ such that $T \models C$ and $\neg\sigma \subseteq C$. An algorithm equipped with a strong $T$-propagate scheme will be called a $\mathsf{CDCL}^*(T)$ solver.

A $\mathsf{CDCL}(T)$ algorithm equipped with a weak $T$-propagation scheme is equivalent to the basic theory propagation rules found in SMT solvers (see, for example, [6,30]). For technical convenience we assume that the weak $T$-propagate scheme adds a clause to the database *certifying* the unit propagation, while in actual implementations the clause would likely not be added and the literal would simply be propagated. Recent SMT solvers[14,12] have strengthened the interaction between the SAT solver and the theory solver, allowing the theory solver to return constraints over new variables; this is modelled very generally by strong $T$-propagate schemes.

## 5.3. Polynomial Equivalence of $\mathsf{CDCL}(T)$ and $\mathsf{Res}(T)$

In the remainder of the section we show that $\mathsf{CDCL}(T)$ and $\mathsf{Res}(T)$ polynomially simulate each other, provided that $\mathsf{CDCL}(T)$ is provided with a (non-deterministic) sequence of variable choices in advance and it has a (i.e. *asserting*) clause learning scheme. The proof presented here is a modification of the seminal result of Pipatsrisawat and Darwiche [34]; the definitions below are modified appropriately for the $\mathsf{CDCL}(T)$ setting.

In the CDCL(T) context, the definitions of assignment trail, extended branching sequence, empowering (dually, absorbed) clauses are very similar to the CDCL SAT solver case. We reproduce these definitions here with appropriate modifications.

Just as in the case of CDCL solvers, in the $\mathsf{CDCL}(T)$ setting an *assignment trail* is a sequence of pairs $\sigma = \{(\ell_1, d_1), (\ell_2, d_2), \ldots, (\ell_t, d_t)\}$ where each literal $\ell_i$ is a literal from the theory and each $d_i \in \{\mathsf{d}, \mathsf{p}\}$, indicating that the literal was set by the solver by a decision or a unit propagation respectively. The *decision level* of a literal $\ell_i$ in the branching sequence is the number of decision literals occurring in $\sigma$ up to and including $\ell_i$. The *state* of a $\mathsf{CDCL}(T)$ solver can be defined as $(\mathscr{F}, \Gamma, \sigma)$, where $\mathscr{F}$ is the input CNF formula, $\Gamma$ is a set of learned clauses, and $\sigma$ is an assignment trail. Given an assignment trail $\sigma$ and a clause $C$ we say that $C$ is *asserting* if it contains exactly one literal occuring in $\sigma$ of decision level $|C|$. A clause learning scheme is *asserting* if all conflict clauses produced by the scheme are asserting with respect to the assignment trail at the time of conflict.

A *extended branching sequence* is an ordered sequence $B = \{\beta_1, \beta_2, \ldots, \beta_t\}$ where each $\beta_i$ is either (1) a literal from the theory, (2) a symbol $x \in \{\mathsf{R}, \mathsf{NR}\}$, to denote a restart or no-restart, respectively, or (3) a clause $C$ such that $T \models C$. If $A$ is a $\mathsf{CDCL}(T)$ solver with a $T$-propagate scheme, we use an extended branching sequence to dictate the operation of the solver $A$ on $\mathscr{F}$: whenever the solver calls the Branching Scheme, we consume the next $\beta_i$ from the sequence. If it is a literal from the theory, then we set that literal; otherwise we halt the execution in error.

Similarly, when the $\mathsf{CDCL}(T)$ solver calls the Restart Scheme we can use the branching sequence to dictate whether or not to restart, and when the solver calls the weak $T$-propagate scheme we use the sequence to dictate which clause to learn. If the symbol does not correctly match the current scheme being called then we halt in error. If the branching sequence is empty, then simply proceed using the heuristics defined by the algorithm. We now introduce a central notion that was originally defined by Pipatsrisawat and Darwiche [34] and separately Atserias, Fichte and Thurley [2].

**Definition 5.2** (Empowering Clauses)**.** Let $\mathscr{F}$ be a collection of clauses over an arbitrary theory $T$ and let $A$ be a $\mathsf{CDCL}(T)$ solver. Let $C = (\alpha \Rightarrow \ell)$ be any clause. We say that $C$ is *empowering with respect to $\mathscr{F}$ at $\ell$* if the following holds: (1) $\mathscr{F} \models C$, (2) $\mathscr{F} \wedge \alpha$ is unit consistent, and (3) any execution of $A$ on $\mathscr{F}$ that falsifies all literals in $\alpha$ does not unit propagate $\ell$. The literal $\ell$ is said to be *empowering*. If item (1), (2) are satisfied but (3) is false then we say that the solver $A$ and $\mathscr{F}$ *absorbs* $C$ at $\ell$; if $A$ and $\mathscr{F}$ absorbs $C$ at at every literal then the clause is simply *absorbed*. (The concepts of empowering and absorbed are duals of each other.)

One should think of the absorbed clauses as being "learned implicitly" — absorbed clauses may not necessarily appear in $\mathscr{F}$. However, if we assign all but one of the literals in the clause to false then unit propagation in $\mathsf{CDCL}(T)$ will set the final literal to true. That is, even if the absorbed clause $C$ is not in $\mathscr{F}$, the unit propagation sub-routine behaves “as though” the absorbed clause is actually in $\mathscr{F}$.

Symmetrically, in order for a clause $C$ to be learned by a $\mathsf{DPLL}$ solver, it must be *empowering at some literal* $\ell$ at the time it is learned. To see this, consider a trace of a $\mathsf{DPLL}$ solver wherein we have just learned a clause $C$. Since we have learned $C$ it easy to see that it must be the case that $\mathscr{F} \models^T C$. Let $\sigma$ be the branching sequence leading to the conflict in which we learned $C$, and let $\ell$ be the last decision literal assigned in $\sigma$

*before* the solver hit a conflict (if DPLL uses an asserting clause learning scheme, such a literal must exist). We can write $C \equiv (\alpha \Rightarrow \neg\ell)$, and clearly $\alpha \subseteq \sigma$. Thus, at the point in the branching sequence $\sigma$ before we assign $\ell$ it must be that $\mathscr{F} \wedge \alpha$ is unit consistent, since we have assigned another literal after assigning each of the literals in $\alpha$. Finally, $\mathscr{F} \wedge \alpha \not\vdash_1^T \ell$ since $\neg\ell$ was chosen as a decision literal *after* we set the literals in $\alpha$.

We first introduce a simple lemma that shows we can construct an extended branching sequence which allows us to learn any theory clause that we like.

**Lemma 5.1.** *Let $\mathscr{F}$ be an unsatisfiable CNF over a theory $T$ and let $\Pi$ be any $\mathsf{Res}(T)$ proof from $\mathscr{F}$. Let $\Pi_T \subseteq \Pi$ be the set of clauses in $\Pi$ derived using the theory rule. For any $\mathsf{CDCL}(T)$ algorithm $A$ there is an extended branching sequence $B$ such that after applying $B$ to the solver $A$ every clause in $\Pi_T$ will be absorbed.*

*Proof.* Order $\Pi_T$ arbitrarily as $C_1, C_2, \ldots, C_m$ and remove any clause that is absorbed or already in $\mathscr{F}$, as these are clearly already absorbed. We construct $B$ directly: query the negations of literals in $C_1$ and when we have queried all but one literal in $C_1$, add the clause $C_1$ to the extended branching sequence. By definition the weak $T$-propagator will be called and will return $C_1$, adding it to the clause database. Restart and continue to the next theory clause in order. □

Our proof of mutual simulations between $\mathsf{Res}(T)$ and $\mathsf{CDCL}(T)$ crucially relies on the following lemma (which is a modified version of a lemma from [34]). We say a clause $C$ is *unit refutable* from $\mathscr{F}$ if $\mathscr{F} \wedge \neg C$ is not unit consistent, i.e. $\mathscr{F} \wedge \neg C \vdash_1 \emptyset$. (This is similar to the 1-provable clause definition in the CDCL SAT solver setting.)

**Lemma 5.2.** *Let $\mathscr{F}$ be an unsatisfiable, unit-consistent CNF over literals from a theory $T$ and let $\Pi$ be any $\mathsf{Res}(T)$ proof from $\mathscr{F}$. Let $\Pi_T$ be the set of clauses in $\Pi$ derived using the theory rule. Then there exists an extended branching sequence $B$ and a clause $C$ in $\Pi$ that is both empowering and unit-refutable with respect to $\mathscr{F} \cup \Pi_T$.*

*Proof.* Let $\Pi$ denote a $\mathsf{Res}(T)$-proof of $\mathscr{F}$, whose clauses we order as $C_1, C_2, \ldots, C_m$. It follows from the assumptions that there exists a $C_i$ which is the first clause in $\Pi$ by this ordering such that it is not unit-refutable. Since $\Pi$ is a $\mathsf{Res}(T)$-proof, $C_i$ is one of three types: either it is a clause in $\mathscr{F}$, it is a clause derived from the theory rule, or $C_i$ was derived by applying the resolution rule to two clauses $C_j, C_k$. If $C_i \in \mathscr{F}$ then it is clearly unit-refutable, which is a contradiction.

If $C_i$ was derived from the theory rule, then observe that all such clauses are absorbed with respect to $\mathscr{F} \cup \Pi_T$.

Finally, suppose that $C_i$ was derived by applying the resolution rule to clauses $C_j$ and $C_k$, and we write $C_j = (\alpha \Rightarrow \ell), \quad C_k = (\beta \Rightarrow \overline{\ell})$ where $\ell$ is the resolved literal, where $j, k < i$ in the ordering of clauses in $\Pi$. Since $C_j$ and $C_k$ are both unit-refutable, we assume (by way of contradiction) that neither $C_j$ nor $C_k$ are empowering. It follows by definition that both clauses are absorbed at every literal. Thus, if we consider $\mathscr{F} \wedge \alpha \wedge \beta$, it follows by the absorption property that $\mathscr{F} \wedge \alpha \wedge \beta \vdash_1 \ell, \mathscr{F} \wedge \alpha \wedge \beta \vdash_1 \neg\ell$ which of course implies that $\mathscr{F} \wedge \alpha \wedge \beta \vdash_1^T \emptyset$. However, $\overline{C}_i = \alpha \wedge \beta$, and so $C_i$ is unit-refutable, which is a contradiction! Thus at least one of $C_j$ or $C_k$ is both empowering and unit-refutable. □

The gist of the lemma 5.2 is simple: if clauses $C \vee \ell$ and $D \vee \overline{\ell}$ are both absorbed by a collection of clauses $\mathscr{C}$, then asserting $\overline{C} \wedge \overline{D}$ in the DPLL solver will hit a conflict since it will unit-imply both $\ell$ and $\overline{\ell}$. Now we are ready to prove the main theorem of this section, which is the difficult direction of showing $\mathsf{Res}(T) \equiv_p \mathsf{CDCL}(T)$. The main step of the theorem is a claim which shows that empowering and unit-refutable clauses will be absorbed by the solver after sufficiently many restarts.

**Theorem 5.3.** *The* $\mathsf{CDCL}(T)$ *system with an asserting clause learning scheme, non-deterministic branching and* $T$*-propagation polynomially simulates* $\mathsf{Res}(T)$*. Equivalently: for any unsatisfiable CNF* $\mathscr{F}$ *over a theory* $T$*, and any* $\mathsf{Res}(T)$ *refutation* $\Pi$ *of* $\mathscr{F}$ *there exists an extended branching sequence* $B$ *such that running a* $\mathsf{CDCL}(T)$ *algorithm on input* $\mathscr{F}$ *using the sequence* $B$ *will refute* $\mathscr{F}$ *in time polynomial in the length of* $|\Pi|$.

*Proof.* Let $\mathscr{F}$ be an unsatisfiable CNF over the theory $T$, and let $\Pi$ be a $\mathsf{Res}(T)$ refutation of $\mathscr{F}$. Let $\Pi_T \subseteq \Pi$ be the set of clauses in $\Pi$ derived using the theory rule, and write $\Pi = C_1, C_2, \ldots, C_m$. As a first step, apply Lemma 5.1 and construct an extended branching sequence $B'$ which leads to the absorbtion of all clauses in $\Pi_T$. We prove the following claim, from which the theorem easily follows.

**Claim.** Let $C$ be any unit-refutable and empowering clause with respect to $\mathscr{F}$. Then there exists an extended branching sequence $B$ of polynomial size such that after applying $B$ the clause $C$ will be absorbed.

Let $\ell$ be any empowering literal of $C$, and write $C = (\alpha \Rightarrow \ell)$. Let $B$ be any extended branching sequence in which all literals in $\alpha$ are assigned. Since $C$ is empowering, it follows that $\mathscr{F} \wedge \alpha$ is unit-consistent. Extending $B$ with the decision literal $\neg\ell$ will therefore cause a conflict since $C$ is unit-refutable. Let $C'$ be the asserting clause obtained by applying the clause learning scheme to $B \cup \{\neg\ell\}$. If $\mathscr{F} \wedge C'$ absorbs $C$ at $\ell$, then we are done and we continue to the next empowering literal. Otherwise, we resolve whatever conflicts the solver needs to resolve (possibly adding more learned clauses along the way) until the branching sequence is unit-consistent.

Observe that after this process we must have that $\mathscr{F} \wedge C' \vdash_1 \ell'$ where $\ell'$ is some literal at the same decision level as $\ell$, since the clause learning scheme is asserting. Thus the number of literals at the maximum decision level has reduced by one. At this point, we restart and do exactly the same sequence of branchings — each time, as argued above, we reduce the number of literals at the maximum decision level by 1. Since $\ell$ is a literal at the maximum decision level, it implies that after at most $O(n)$ restarts (and $O(n)$ learned clauses) we will have absorbed the clause $C$ at $\ell$. Repeating this process at most $n$ times for each empowering literal in $C$ we can absorb $C$, and it is clear that the number of learned clauses is polynomial from the analysis. We are now ready to finish the proof.

Apply the claim repeatedly to the first empowering and unit-refutable clause in $\Pi$ to absorb that clause — by Lemma 5.2, such a clause will exist as long as the CNF $\mathscr{F}$ is not unit-refutable; a $\mathsf{CDCL}(T)$ solver can obtain an arbitrary theory clause by setting relevant literals in the branching sequence and using theory propagation. Since the length of the proof $\Pi$ is finite (length $m$), it follows that this process must terminate after at most $m$ iterations. At this point, there can not be such an empowering and unit-refutable clause, and so by Lemma 5.2 it follows that $\mathscr{F}$ (with its learned clauses) is now unit-refutable, at which point the $\mathsf{CDCL}(T)$ algorithm halts and outputs UNSAT. $\square$

The reverse direction of the theorem is straightforward, and thus we have the following corollary:

**Corollary 5.3.1.** *The* $\mathsf{CDCL}(T)$ *system with an asserting clause learning scheme, non-deterministic branching and* $T$*-propagation is polynomially equivalent to* $\mathsf{Res}(T)$.

A key point of the above simulation is that it does not depend on whether or not the $T$-propagation scheme is weak or strong — since the clauses learned by the scheme are specified in advance by the extended branching sequence the same proof will apply directly if we began with a $\mathsf{Res}^*(T)$ proof instead. Of course, if we begin with a $\mathsf{Res}^*(T)$ proof instead of a $\mathsf{Res}(T)$ proof we may use the full power of the theory derivation rule, requiring that we use a $\mathsf{DPLL}^*(T)$ algorithm with a strong $T$-propagation scheme instead. We record this observation as a second theorem.

**Theorem 5.4.** *The* $\mathsf{DPLL}^*(T)$ *system with an asserting clause learning scheme, non-deterministic branching and* $T$*-propagation is polynomially equivalent to* $\mathsf{Res}^*(T)$.

In conclusion, we establish here that the CDCL(T) architecture is best understood through the lens of $\mathsf{Res}(T)$ and $\mathsf{Res}^*(T)$ proof system. This kind of proof complexity theoretic analysis enables to appropriately characterize the power of these systems.

### *5.4.* First-order Decision Procedures and their Combinations

A considerable amount of research on solvers for *individual* first-order theories and their combinations has been conducted and many practically efficient algorithms for these theories have been developed (See the chapter on SMT solvers in the Handbook of Satisfiability [4]). As we discussed earlier, in these notes we deliberately focus on the $\mathsf{CDCL}(T)$ architecture and its proof complexity-theoretic analysis, complementing other surveys that discuss algorithms for deciding the satisfiability problem for individual first-order theories and the idea of combinations of decision procedures in detail.

## 6. Lessons Learnt and Conclusions

In these notes, we have provided a detailed description of the inner workings of SAT and SMT solvers, and we have done so through the lens of "solvers as proof systems" perspective. We described two sets of result to bolster this viewpoint. First, we described the breakthrough result by Pipatsrisawat and Darwiche [34], which for the first time established the link between CDCL and general resolution via polynomial simulation of proof systems. The value of this result is that it explains how clause learning is the key heuristic that gives CDCL solvers their power relative to DPLL, which are polynomially equivalent to the much weaker tree-like resolution. We also showcased the result of Robere et al. [35], that generalizes the "CDCL is polynomially equivalent resolution" proof to the SMT case ($\mathsf{CDCL}(T)$ is equivalent to $\mathsf{Res}(T)$, and when new variables are allowed, it is equivalent to $\mathsf{Res}^*(T)$). Once again, this enables us to really identify where the power of these solvers come from.

The second advantage of the solvers as proof systems model is that it enables us to view solver implementations as proof search algorithms which consist of subroutines that correspond to proof rules and methods for optimal sequencing and selection of these rules. This suggests that the problem of designing solver algorithms can be recast as coming up with appropriate proof rules and optimization procedures to adaptively sequence/select them for classes of instances. Given that current solvers produce copious amounts of data as they search for proofs and/or solutions, it further suggests that such optimization procedures may best be designed by leveraging machine learning algorithms[22]. We showcased the work on machine learning based variable selection heuristics, a la, Learning Rate-based Branching (LRB)[22] and machine learning-based restarts[26]. Many researchers are now following up on this idea. Particularly notworthy is the work of Mate Soos and Kuldeep Meel in developing a machine learing based SAT framework that would enable researchers to develop their own machine learning-based SAT heuristics.

These are powerful ideas, and a whole host of researchers are pursuing deeper understanding of solvers via proof complexity and machine learning. These ideas are now being applied to mixed integer solvers, MaxSAT solvers, QBF solvers, parallel versions of SAT and SMT solvers, and entire classes of automated reasoning algorithms that is beyond the scope of these notes. The foundational perspective presented here reinforces the point that solver research is not merely a jumble of heuristics developed in response to applications, but rather a deep set of ideas with far-reaching impact.

## References

[1] Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. Counterexample guided inductive synthesis modulo theories. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, pages 270–288, 2018.

[2] Albert Atserias, Johannes Klaus Fichte, and Marc Thurley. Clause-learning algorithms with many restarts and bounded-width resolution. *J. Artif. Intell. Res.*, 40:353–373, 2011.

[3] Paul Beame, Henry A. Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *J. Artif. Intell. Res.*, 22:319–351, 2004.

[4] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[5] Nikolaj Bjørner, Bruno Dutertre, and Leonardo de Moura. Accelerating lemma learning using joins - DPLL(Join). In *15th International Conference on Logic for Programming Artificial Intelligence and Reasoning, LPAR'08*, 2008.

[6] Nikolaj Bjørner and Leonardo De Moura. Tractability and modern SMT Solvers. In L. Bordeaux, Y. Hamadi, and P. Kohli, editors, *Tractability: Practical Approaches to Hard Problems*, pages 350–377. Cambridge University Press, 2014.

[7] Aaron R. Bradley. SAT-based Model Checking Without Unrolling. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'11, pages 70–87, Berlin, Heidelberg, 2011. Springer-Verlag.

[8] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM.

[9] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[10] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[11] Martin Davis, George Logemann, and Donald Loveland. A Machine Program for Theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[12] Leonardo De Moura and Nikolaj Bjørner. The Z3 Theorem Prover. https://github.com/Z3Prover, 2008.

[13] Peter J Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *Journal of the ACM (JACM)*, 27(4):758–771, 1980.

[14] Bruno Dutertre and Leonardo De Moura. The Yices SMT Solver. `http://yices.csl.sri.com/`, 2006.

[15] Niklas Eén and Niklas Sörensson. *Theory and Applications of Satisfiability Testing: 6th International Conference, SAT 2003, Santa Margherita Ligure, Italy, May 5-8, 2003, Selected Revised Papers*, chapter An Extensible SAT-solver, pages 502–518. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[16] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 519–531. Springer, 2007.

[17] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Dpll (t): Fast decision procedures. In *International Conference on Computer Aided Verification*, pages 175–188. Springer, 2004.

[18] Carla P. Gomes, Bart Selman, Nuno Crato, and Henry Kautz. Heavy-Tailed Phenomena in Satisfiability and Constraint Satisfaction Problems. *Journal of Automated Reasoning*, 24(1-2):67–100, February 2000.

[19] Daniel Jackson. Alloy: A Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, April 2002.

[20] Stasys Jukna. *Boolean Function Complexity: Advances and Frontiers*. Springer Publishing Company, Incorporated, 2012.

[21] Hadi Katebi, Karem A. Sakallah, and João P. Marques-Silva. Empirical Study of the Anatomy of Modern Sat Solvers. In *Proceedings of the 14th International Conference on Theory and Application of Satisfiability Testing*, SAT'11, pages 343–356, Berlin, Heidelberg, 2011. Springer-Verlag.

[22] Jia Hui Liang. *Machine Learning for SAT Solvers*. PhD thesis, University of Waterloo, Canada, 2018.

[23] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Exponential Recency Weighted Average Branching Heuristic for SAT Solvers. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 3434–3440. AAAI Press, 2016.

[24] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning Rate Based Branching Heuristic for SAT Solvers. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing – SAT 2016*, pages 123–140, Cham, 2016. Springer International Publishing.

[25] Jia Hui Liang, Vijay Ganesh, Ed Zulkoski, Atulan Zaman, and Krzysztof Czarnecki. Understanding VSIDS Branching Heuristics inÂăConflict-Driven Clause-Learning SAT Solvers. In Nir Piterman, editor, *Hardware and Software: Verification and Testing*, pages 225–241, Cham, 2015. Springer International Publishing.

[26] Jia Hui Liang, Chanseok Oh, Minu Mathew, Ciza Thomas, Chunxiao Li, and Vijay Ganesh. Machine learning-based restart policy for CDCL SAT solvers. In *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, pages 94–110, 2018.

[27] João P Marques-Silva and Karem A. Sakallah. GRASP-A New Search Algorithm for Satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '96, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society.

[28] Matthew W. Moskewicz, Conor F. Madigan, and Sharad Malik. Method and system for efficient implementation of boolean satisfiability, August 26 2008. US Patent 7,418,369.

[29] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.

[30] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories. *Journal of the ACM*, 53(6):937–977, nov 2006.

[31] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). *Journal of the ACM*, 53(6):937–977, 2006.

[32] Albert Oliveras and Enric RodrÄśguez-Carbonell. Combining Decision Procedures : The Nelson-Oppen approach. *Techniques*, 2009.

[33] Knot Pipatsrisawat and Adnan Darwiche. A Lightweight Component Caching Scheme for Satisfiability Solvers. In *Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing*, SAT'07, pages 294–299, Berlin, Heidelberg, 2007. Springer-Verlag.

[34] Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning SAT solvers as resolution engines. *Artif. Intell.*, 175(2):512–525, 2011.

[35] Robert Robere, Antonina Kolokolova, and Vijay Ganesh. The proof complexity of SMT solvers. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, pages 275–293, 2018.

[36] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT Solvers to Cryptographic Problems. In *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, pages 244–257, 2009.

[37] G. S. Tseitin. *On the Complexity of Derivation in Propositional Calculus*, pages 466–483. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983.

*Engineering Secure and Dependable Software Systems*
*A. Pretschner et al. (Eds.)*
*IOS Press, 2019*
*© 2019 The authors and IOS Press. All rights reserved.*
*doi:10.3233/978-1-61499-977-5-61*

# Programming by Examples: PL Meets ML

Sumit Gulwani [a], Prateek Jain [b]
[a] *Microsoft Corporation, Redmond, USA*
[b] *Microsoft Research, Bangalore, India*

**Abstract.** Programming by Examples (PBE) involves synthesizing intended programs in an underlying domain-specific programming language (DSL) from example-based specifications. This new frontier in AI enables computer users, 99% of whom are non-programmers, to create scripts to automate repetitive tasks. PBE can provide 10-100x productivity increase for data scientists, business users, and developers for various task domains like string/number/date transformations, structured table extraction from log files/web pages/PDF/semi-structured spreadsheets, transforming JSON from one format to another, repetitive text editing, repetitive code refactoring and formatting. PBE capabilities can be surfaced using GUI-based tools, code editors, or notebooks, and the code can be synthesized in various target languages like Java or even PySpark to facilitate efficient execution on big data.

There are three key components in a PBE system. (i) A search algorithm that can efficiently search for programs that are consistent with the examples provided by the user. We leverage a divide-and-conquer based deductive search paradigm that inductively reduces the problem of synthesizing a program expression of a certain kind that satisfies a given specification into sub-problems that refer to sub-expressions or sub-specifications. (ii) Program ranking techniques to pick an intended program from among the many that satisfy the examples provided by the user. (iii) User interaction models to facilitate usability and debuggability.

Each of these PBE components leverage both symbolic reasoning and heuristics. We make the case for synthesizing these heuristics from training data using appropriate machine learning methods. In particular, we use neural-guided heuristics to resolve any resulting non-determinism in the search process. Similarly, our ML-based ranking techniques, which leverage features of program structure and program outputs, are often able to select an intended program from among the many that satisfy the examples. Finally, Our active-learning-based user interaction models, which leverage clustering of input data and semantic differences between multiple synthesized programs, facilitate a bot-like conversation with the user to aid usability and debuggability. That is our algorithms that deeply integrate neural techniques with symbolic computation can not only lead to better heuristics, but can also enable easier development, maintenance, and even personalization of a PBE system.

**Keywords.** Program synthesis, Programming by Examples, Search algorithm, Program ranking, Active learning, Data wrangling

[0]This is an extended version of the article with the same title that appeared in the proceedings for APLAS 2017 as an invited-talk contribution and was published by Springer [13]. This revision includes more than 5 pages of new content, which includes new figures and references along with expansion and better phrasing of some earlier content.

## 1. Introduction

Program Synthesis is the task of synthesizing a program that satisfies a given specification [15]. The traditional view of program synthesis has been to synthesize programs from logical specifications that relate the inputs and outputs of the program. A typical academic exercise in program synthesis is to synthesize complicated algorithms such as sorting algorithms [43], graph algorithms [18], and bitvector algorithms [19]. For instance, the logical specification for a sorting algorithm would state that the sorting algorithm takes as input an array $A[1 :: n]$ and outputs another array $B[1 :: n]$ s.t. $B$ is a permutation of $A$, and $B$ is sorted, i.e.,

$$\forall 1 \leq i < n : B[i] \leq B[i+1] \;\wedge$$
$$\exists \sigma, \text{ a permutation of } 1 \ldots n \text{ such that } \forall 1 \leq i < n : B[i] = A[\sigma(i)]$$

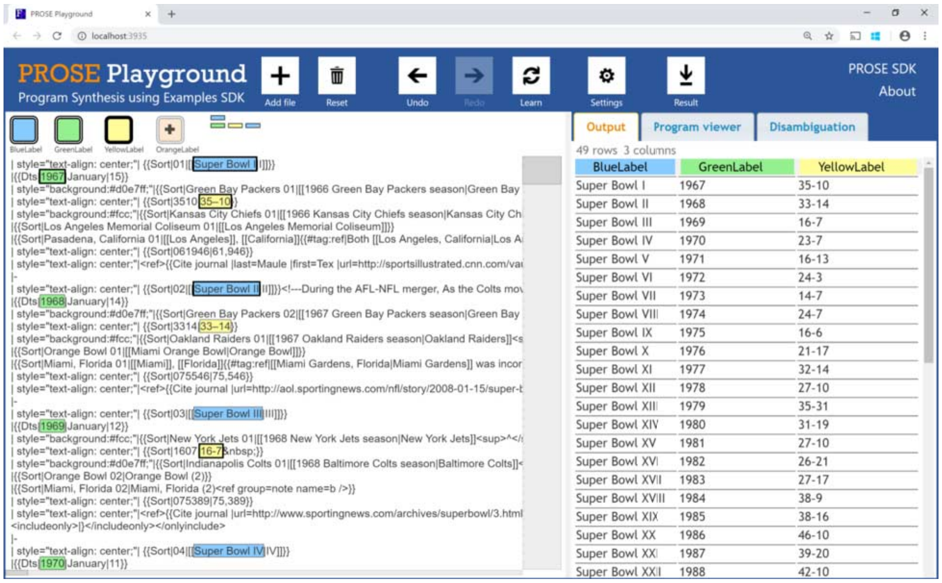
Programming by Examples (PBE) is a sub-field of program synthesis, where the specification consists of input-output examples, or more generally, output properties over given input states [11]. PBE has emerged as a favorable paradigm for two key reasons: (i) the example-based specification in PBE makes PBE more tractable than general program synthesis because it involves reasoning over concrete program states (Section 4 discusses the underlying search techniques). As a result, we can synthesize more complicated and larger programs than what was possible earlier, and we can do that very efficiently and often in real-time to facilitate usability. (ii) Example-based specifications are much easier for the users to provide in many scenarios. This not only increases the usability of synthesis technologies for developers, but also broadens the applicability to end users. This is highly significant since 99% of computer users do not know programming and would find it extremely difficult to write down logical specifications.

The advantages of PBE also present some unique challenges. First, examples are highly ambiguous form of user's intent. There are too many programs that match a small number of examples. Requiring the user to provide a large number of examples to narrow down the ambiguity affects the usability of such systems. We discuss (in Section 5) how program ranking techniques can be designed to guess an intended program from among the many that satisfy a few representative user-provided examples. Secondly, we need technologies that can help the user identify representative inputs on which to provide examples. We discuss this in Section 6.

This article is organized as follows. Section 2 discusses some key applications of PBE. In Section 3, we provide our perspectives on how PBE differs from Machine Learning (ML), both of which aim to learn from examples. We make the case that instead of thinking about ML and PBE as alternatives, ML can actually be used to create better PBE systems. The next few sections discuss opportunities for such an integration for each of the three key components of PBE: search algorithm (Section 4), ranking (Section 5), and interactivity (Section 6). Section 7 presents some directions for future work.

## 2. Applications

The two killer applications for programming by examples today are in the space of data transformations/wrangling and code transformations.



```
| style="text-align: center;"| {{Sort|01|[[Super Bowl I|I]]}}
|{{Dts|1967|January|15}}
| style="background:#d0e7ff;"|{{Sort|Green Bay Packers 01|[[1966 Green Bay Packers season|Green Bay
| style="text-align: center;"| {{Sort|3510|35–10}}
| style="background:#fcc;"|{{Sort|Kansas City Chiefs 01|[[1966 Kansas City Chiefs season|Kansas City Ch
|{{Sort|Los Angeles Memorial Coliseum 01|[[Los Angeles Memorial Coliseum]]}}
|{{Sort|Pasadena, California 01|[[Los Angeles]], [[California]]{{#tag:ref|Both [[Los Angeles, California|Los A
| style="text-align: center;"| {{Sort|061946|61,946}}
| style="text-align: center;"|<ref>{{Cite journal |last=Maule |first=Tex |url=http://sportsillustrated.cnn.com/va
|-
| style="text-align: center;"| {{Sort|02|[[Super Bowl II|II]]}}<!---During the AFL-NFL merger, As the Colts mov
|{{Dts|1968|January|14}}
| style="background:#d0e7ff;"|{{Sort|Green Bay Packers 02|[[1967 Green Bay Packers season|Green Bay
| style="text-align: center;"| {{Sort|3314|33–14}}
| style="background:#fcc;"|{{Sort|Oakland Raiders 01|[[1967 Oakland Raiders season|Oakland Raiders]]<s
|{{Sort|Orange Bowl 01|[[Miami Orange Bowl|Orange Bowl]]}}
|{{Sort|Miami, Florida 01|[[Miami]], [[Florida]]{{#tag:ref|[[Miami Gardens, Florida|Miami Gardens]] was incor
| style="text-align: center;"| {{Sort|075546|75,546}}
| style="text-align: center;"|<ref>{{Cite journal |url=http://aol.sportingnews.com/nfl/story/2008-01-15/super-b
|-
| style="text-align: center;"| {{Sort|03|[[Super Bowl III|III]]}}
|{{Dts|1969|January|12}}
| style="background:#fcc;"|{{Sort|New York Jets 01|[[1968 New York Jets season|New York Jets]]<sup>^</
| style="text-align: center;"| {{Sort|1607|16-7&nbsp;}}
| style="background:#d0e7ff;"|{{Sort|Indianapolis Colts 01|[[1968 Baltimore Colts season|Baltimore Colts]]<
|{{Sort|Orange Bowl 02|Orange Bowl (2)}}
|{{Sort|Miami, Florida 02|Miami, Florida (2)<ref group=note name=b />}}
| style="text-align: center;"| {{Sort|075389|75,389}}
| style="text-align: center;"|<ref>{{Cite journal |url=http://www.sportingnews.com/archives/superbowl/3.html
<includeonly>|}</includeonly></onlyinclude>
|-
| style="text-align: center;"| {{Sort|04|[[Super Bowl IV|IV]]}}
|{{Dts|1970|January|11}}
```

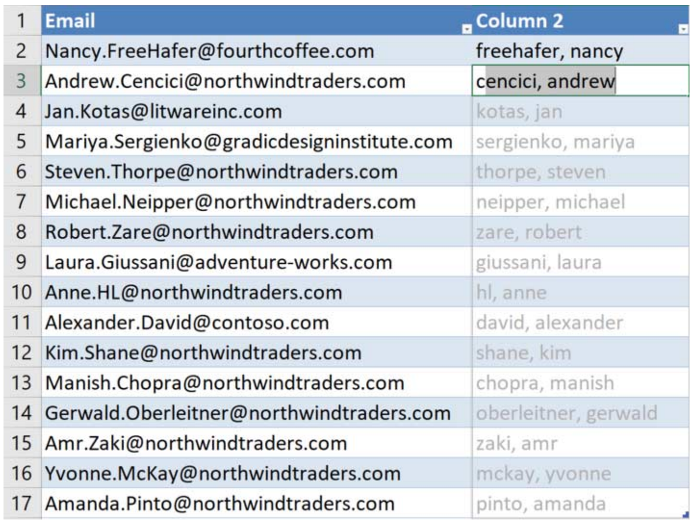
| BlueLabel | GreenLabel | YellowLabel |
|---|---|---|
| Super Bowl I | 1967 | 35-10 |
| Super Bowl II | 1968 | 33-14 |
| Super Bowl III | 1969 | 16-7 |
| Super Bowl IV | 1970 | 23-7 |
| Super Bowl V | 1971 | 16-13 |
| Super Bowl VI | 1972 | 24-3 |
| Super Bowl VII | 1973 | 14-7 |
| Super Bowl VIII | 1974 | 24-7 |
| Super Bowl IX | 1975 | 16-6 |
| Super Bowl X | 1976 | 21-17 |
| Super Bowl XI | 1977 | 32-14 |
| Super Bowl XII | 1978 | 27-10 |
| Super Bowl XIII | 1979 | 35-31 |
| Super Bowl XIV | 1980 | 31-19 |
| Super Bowl XV | 1981 | 27-10 |
| Super Bowl XVI | 1982 | 26-21 |
| Super Bowl XVII | 1983 | 27-17 |
| Super Bowl XVIII | 1984 | 38-9 |
| Super Bowl XIX | 1985 | 38-16 |
| Super Bowl XX | 1986 | 46-10 |
| Super Bowl XXI | 1987 | 39-20 |
| Super Bowl XXII | 1988 | 42-10 |

**Figure 1.** Consider the task of extracting a structured table (shown on the right side) from the custom text file (shown on the left side). This would typically require writing a complicated parsing script involving regular expressions. In contrast, the FlashExtract PBE technology [22] allows automation of such tasks from few examples. This figure illustrates a user experience around this PBE technology. Once the user highlights few examples (often one or two) of a field (using a color unique to that field), FlashExtract synthesizes a program and executes it to extract the other instances and arranges them in a new column in the output table.

## *2.1. Data wrangling*

Data Wrangling refers to the process of transforming the data from its raw format to a more structured format that is amenable to analysis and visualization. It is estimated that data scientists spend 80% of their time wrangling data. Data is locked up into documents of various types such as text/log files, semi-structured spreadsheets, webpages, JSON/XML, and PDF documents. These documents offer their creators great flexibility in storing and organizing hierarchical data by combining presentation/formatting with the underlying data. However, this makes it extremely hard to extract the underlying data for several tasks such as processing, querying, altering the presentation view, or transforming data to another storage format. PBE can make data wrangling easier and faster.

*Extraction:* A first step in a data wrangling pipeline is often that of ingesting or extracting tabular data from semi-structured formats such as text/log files, web pages, and XML/JSON documents. These documents offer their creators great flexibility in storing and organizing hierarchical data by combining presentation/formatting with the underlying data. However, this makes it extremely hard to extract the relevant data. The FlashExtract PBE technology allows extracting structured (tabular or hierarchical) data out of semi-structured documents from examples [22]. For each field in the output data schema, the user provides positive/negative instances of that field and FlashExtract generates a program to extract all instances of that field. The FlashExtract technology ships as the ConvertFrom-String cmdlet in Powershell in Windows 10, wherein the user provides examples of the strings to be extracted by inserting tags around them in test. The FlashEx-



| 1 | Email | Column 2 |
|---|---|---|
| 2 | Nancy.FreeHafer@fourthcoffee.com | freehafer, nancy |
| 3 | Andrew.Cencici@northwindtraders.com | cencici, andrew |
| 4 | Jan.Kotas@litwareinc.com | kotas, jan |
| 5 | Mariya.Sergienko@gradicdesigninstitute.com | sergienko, mariya |
| 6 | Steven.Thorpe@northwindtraders.com | thorpe, steven |
| 7 | Michael.Neipper@northwindtraders.com | neipper, michael |
| 8 | Robert.Zare@northwindtraders.com | zare, robert |
| 9 | Laura.Giussani@adventure-works.com | giussani, laura |
| 10 | Anne.HL@northwindtraders.com | hl, anne |
| 11 | Alexander.David@contoso.com | david, alexander |
| 12 | Kim.Shane@northwindtraders.com | shane, kim |
| 13 | Manish.Chopra@northwindtraders.com | chopra, manish |
| 14 | Gerwald.Oberleitner@northwindtraders.com | oberleitner, gerwald |
| 15 | Amr.Zaki@northwindtraders.com | zaki, amr |
| 16 | Yvonne.McKay@northwindtraders.com | mckay, yvonne |
| 17 | Amanda.Pinto@northwindtraders.com | pinto, amanda |

**Figure 2.** Consider the collection of email addresses in the first column. Suppose the user wants to extract last name and first name and format them as illustrated in the second column. The Flash Fill feature [10] in Excel 2013 (and onwards) allows automation of such repetitive string transformations from few examples. In this case, once the user performs one instance of the desired transformation (row 2, column 2) and proceeds to transforming another instance (row 3, column 2), Flash Fill learns a program concat(ToLower(substr(v,WordToken,12), conststr(", "), ToLower(substr(v,WordToken,1))) that extracts the first two words in input string v (first column), converts them to lowercase, and concatenates them separated by a comma and space.

| | A | B | C | D | E | . . . | R |
|---|---|---|---|---|---|---|---|
| **1** | | value | year | value | year | | Comments |
| **2** | Albania | 1,000 | 1950 | 930 | 1981 | | FRA 1 |
| **3** | Austria | 3,139 | 1951 | 3,177 | 1955 | | FRA 3 |
| **4** | Belgium | 541 | 1947 | 601 | 1950 | | |
| **5** | Bulgaria | 2,964 | 1947 | 3,259 | 1958 | | FRA 1 |
| **6** | Czech . . . | 2,416 | 1950 | 2,503 | 1960 | | NC |

. . .

(a) **Input:** Semi-structured spreadsheet

| | A | B | C | D |
|---|---|---|---|---|
| **1** | Albania | 1,000 | 1950 | FRA 1 |
| **2** | Albania | 930 | 1981 | FRA 1 |

. . .

| | | | | |
|---|---|---|---|---|
| **5** | Austria | 3,139 | 1951 | FRA 3 |
| **6** | Austria | 3,177 | 1955 | FRA 3 |

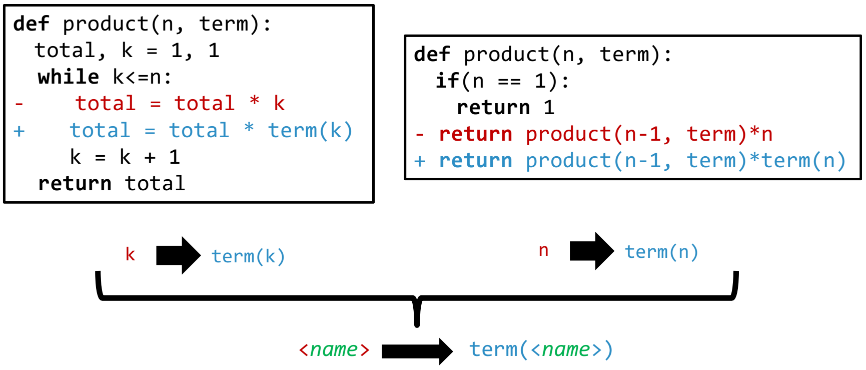
. . .

| | | | | |
|---|---|---|---|---|
| **9** | Belgium | 541 | 1947 | |
| **10** | Belgium | 601 | 1950 | |

. . .

(b) **Output:** Relational table

**Figure 3.** Consider the task of extracting a relational table (b) from the semi-structured spreadsheet (a). The FlashRelate technology [4] allows automation of such tasks from few examples. In this scenario, once the user provides a couple of examples of tuples in the output table (for instance, the highlighted ones), FlashRelate synthesizes a script and executes that script to extract other similar tuples from the input spreadsheet.



```
def product(n, term):
  total, k = 1, 1
  while k<=n:
-    total = total * k
+    total = total * term(k)
     k = k + 1
  return total
```

```
def product(n, term):
  if(n == 1):
    return 1
- return product(n-1, term)*n
+ return product(n-1, term)*term(n)
```

**Figure 4.** This figure shows two incorrect student attempts to a programming problem with a similar kind of fault, wherein the student missed applying the term function. The incorrect statement in each attempt is colored red while the teacher's correction is shown in blue. The Refazer tool [33] can generalize such similar teacher corrections to a more general rule that can be applied to automatically fix other students' attempts with a similar fault.

tract technology also ships in Azure OMS (Operations Management Suite), where it enables extraction of custom fields from log files. Figure 1 illustrates use of this technology to extract structured tabular data from a text file with custom format.

*Transformation:* The Flash Fill feature, released in Excel 2013 and beyond, is a PBE technology for automating syntactic string transformations, such as converting "First-Name LastName" into "LastName, FirstName" [10]. Figure 2 provides an illustration of the Flash Fill feature. PBE can also facilitate more sophisticated string transformations that require lookup into other tables [36]. PBE is also a very natural fit for automating transformations of other data types such as numbers [37] and dates [39].

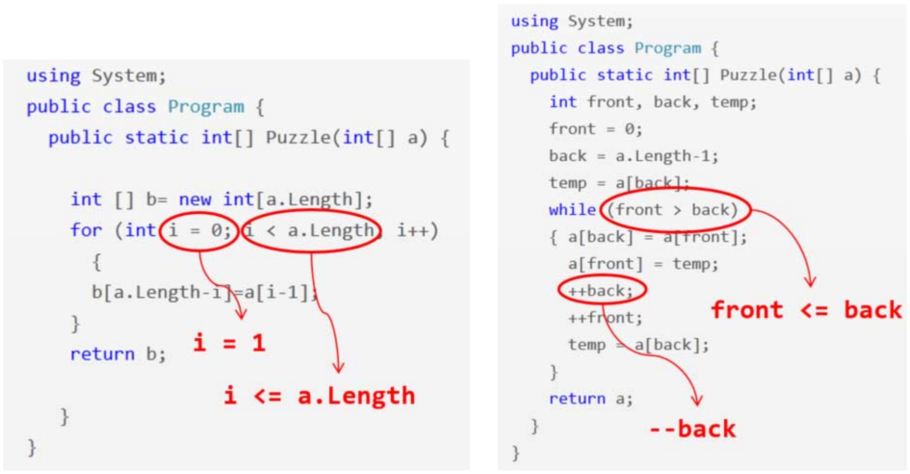
*Formatting:* Another useful application of PBE is in the space of formatting data tables. This can be useful to convert semi-structured tables found commonly in spreadsheets into proper relational tables [4], or for re-pivoting the underlying hierarchical data that has been locked into a two-dimensional tabular format [16]. Figure 3 provides illustrates use of a PBE technology for performing example-based formatting. PBE can also be useful in automating repetitive formatting in a PowerPoint slide deck such as converting all red colored text into green, or switching the direction of all horizontal arrows [31].

## 2.2. *Code Transformations*

There are several situations where repetitive code transformations need to be performed and examples can be used to automate this tedious task.

A standard scenario is that of general code refactoring. As software evolves, developers edit program source code to add features, fix bugs, or refactor it for readability, modularity, or performance improvements. For instance, to apply an API update, a developer needs to locate all references to the old API and consistently replace them with the new API. Examples can be used to infer such edits from a few examples [33].

Another important scenario is that of *application migration*—whether it is about moving from on-prem to the cloud, or from one framework to another, or simply moving



```
using System;
public class Program {
  public static int[] Puzzle(int[] a) {

    int [] b= new int[a.Length];
    for (int i = 0; i < a.Length; i++)
      {
      b[a.Length-i]=a[i-1];
    }
    return b;

  }
}
```

```
using System;
public class Program {
  public static int[] Puzzle(int[] a) {
    int front, back, temp;
    front = 0;
    back = a.Length-1;
    temp = a[back];
    while (front > back)
    { a[back] = a[front];
      a[front] = temp;
      ++back;
      ++front;
      temp = a[back];
    }
    return a;
  }
}
```

**Figure 5.** This figure shows two incorrect student attempts to the problem of reversing an array. The Auto-Grader tool [40] can find small edits to an incorrect attempt (shown in red) that transforms the program into a version that satisfies a given reference set of test cases.

from an old version of a framework to a newer version to keep up with the march of technology. A significant effort is spent in performing repetitive edits to the underlying application code. In particular, for database migration, it is estimated that up to 40% of the developer effort can be spent in performing repetitive code changes in the application code.

Yet another interesting scenario is in the space of feedback generation for programming assignments in programming courses. For large classes such as massive open online courses (MOOCs), manually providing feedback to different students is an unfeasible burden on the teaching staff. We observe that student submissions that exhibit the same fault often need similar fixes. The PBE technology can be used to learn the common fixes from corrections made by teachers on few assignments, and then infer application of these fixes to the remaining assignments, forming basis for automatic feedback [33]. Figure 4 illustrates such a use case. Another possibility is to search for a set of small edits to the student's incorrect attempt to make it pass a reference set of test cases, as illustrated in Figure 5.

## 3. PL meets ML

It is interesting to compare PBE with Machine learning (ML) since both involve example-based training and prediction on new unseen data. PBE learns from very few examples, while ML typically requires large amount of training data. The models generated by PBE are human-readable (in fact, editable programs) unlike many black-box models produced by ML. PBE generates small scripts that are supposed to work with perfect precision on any new valid input, while ML can generate sophisticated models that can achieve high, but not necessarily perfect, precision on new varied inputs. Hence, given their complementary strengths, we believe that PBE is better suited for relatively simple well-defined tasks, while ML is better suited for sophisticated and fuzzy tasks.

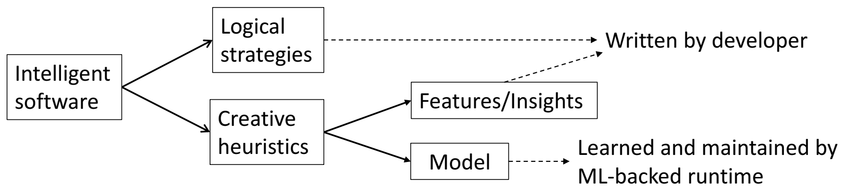


**Figure 6.** A proposal for development of intelligent software that facilitates increased developer productivity and increased software intelligence.

Recently, *neural program induction* has been proposed as a fully ML-based alternative to PBE. These techniques develop new neural architectures that learn how to generate outputs for new inputs by using a latent program representation induced by learning some form of neural controller. Various forms of neural controllers have been proposed such as ones that have the ability to read/write to external memory tape [9], stack augmented neural controller [20], or even neural networks augmented with basic arithmetic and logic operations [27]. These approaches typically involve developing a continuous representation of the atomic operations of the network, and then using end-to-end training of a neural controller or reinforcement learning to learn the program behavior. While this is impressive, these techniques aren't a good fit for the PBE task domains of relatively simple well-defined tasks. This is because these techniques don't generate an interpretable model of the learned program, and typically require large computational resources and several thousands of input-output examples per synthesis task. We believe that a big opportunity awaits in carefully combining ML-based data-driven techniques with Programming Languages (PL)-based logical reasoning approaches to improve a standard PBE system as opposed to replacing it.

## 3.1. A perspective on PL meets ML

AI software often contains two intermingled parts: logical strategies + creative heuristics. Heuristics are difficult to author, debug, and maintain. Heuristics can be decomposed into two parts: insights/features + model/scoring function over those features. We propose that an AI developer refactors their intelligent code into logical strategies and declarative features while ML techniques are used to evolve an ideal model or scoring function over those insights with continued feedback from usage of the intelligent software. This has two advantages: (i) Increase in developers productivity, (ii) Increase in systems intelligence because of better heuristics and those that can adapt differently to different workloads or unpredictable environments (a statically fixed heuristic cannot achieve this).

Figure 6 illustrates this proposed modular construction of intelligent software. Developing an ML model in this framework (where the developer authors logical strategies and declarative insights) poses several interesting open questions as traditional ML techniques are not well-equipped to handle such declarative and symbolic frameworks. Moreover, even the boundary between declarative insights and ML-based models may be

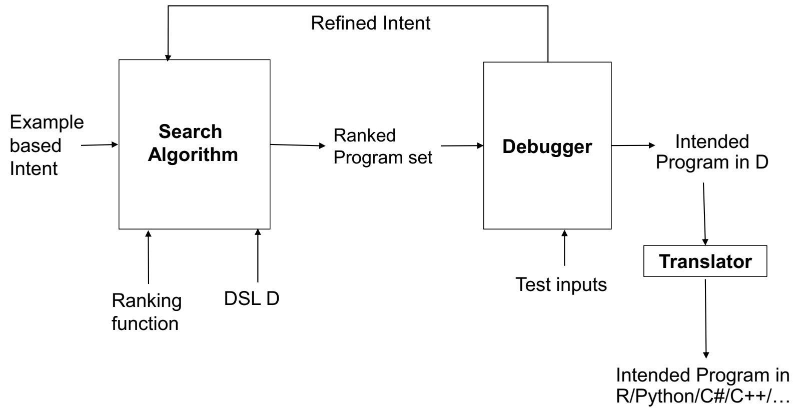


**Figure 7.** Programming-by-Examples Architecture. The search algorithm, parameterized by a domain-specific language (DSL) and a ranking function, synthesizes a ranked set of programs from the underlying DSL that are consistent with the examples provided by the user. The debugging component, which leverages additional test inputs, interacts with the user to refine the specification and the synthesis process is repeated. Once an intended program has been synthesized, it can be translated to a target language using standard syntax-directed translation.

fluid. Depending on the exact problem setting as well as the domain, the developer might want to decide which part of the system should follow deterministic logical reasoning and which part should be based on data-driven techniques.

### *3.2. Using ML to improve PBE*

There are three key components in a PBE engine: search algorithm, ranking strategy, and user interaction models. Each of these components leverage various forms of heuristics. ML can be used to learn these heuristics, thereby improving the effectiveness and maintainability of the various PBE components. In particular, ML can be used to speed up the search process by predicting the success likelihood of various paths in the huge search space [21]. It can be used to learn a better ranking function [26]. It can be used to cluster test data and associate confidence measure over the outputs generated by the synthesized program to drive an effective active learning session with the user for debuggability [28].

## 4. Search Algorithm

Figure 7 shows the architecture of a PBE system. The most involved technical component is the search algorithm, which we discuss in this section. Section 4.1 and 4.2 describe the two key ingredients that form the foundation for designing this search algorithm. These ingredients are based on deterministic logical reasoning. Section 4.3 then discusses and speculates how machine learning can further help exploit the traditional PL-driven logical reasoning to obtain an even more efficient, real-time search algorithm for PBE.

String Expression E:=concat($E_1$, $E_2$) | substr(E, $P_1$, $P_2$) | conststr(*String*)
Position P:= *Integer* | pos(x, $R_1$, $R_2$, k)

**Figure 8.** An example Domain Specific Language (DSL).substr, concat are operators to manipulate the string and conststr represents a constant string. pos operator identifies position of a particular pattern in the input x. *String* is any constant string and *Integer* is an arbitrary integer that can be negative as well.

### 4.1. Domain-specific Language

A key idea in program synthesis is to restrict the search space to an underlying domain-specific language (DSL) [1,12]. The DSL should be expressive enough to represent a wide variety of tasks in the underlying task domain, but also restricted enough to allow efficient search. We have designed many functional domain-specific languages for this purpose, each of which is characterized by a set of operators and a syntactic restriction on how those operators can be composed with each other (as opposed to allowing all possible type-safe composition of those operators) [11]. A DSL is typically specified as a context-free grammar that consists of one or more production rules for each non-terminal. The right hand side of a production rule can be either another non-terminal or an explicit set of program expressions or a program operator applied to some non-terminals.

For illustration, we present an extremely simple string manipulation grammar in Figure 8; this DSL is a heavily stripped down version of Flash Fill DSL [10]. The language has two key operators for string manipulations: a) substr operator which takes as input a string x, and two position expressions $P_1$ and $P_2$ that evaluate to positions/indices within the string x, and returns the substring between those positions, b) concat which concatenates the given expressions. The choice for position expression P includes the pos(x,$R_1$,$R_2$,k) operator, which returns the $k^{th}$ position within the string x such that (some suffix of) the left side of that position matches with regular expression $R_1$ and (some prefix of) the right side of that position matches with regular expression $R_2$.

For example, program given by,
concat(substr(Input, $\varepsilon$, " ", 1), substr(Input, " ", $\varepsilon$, -1), conststr("@cs.colorado.edu"))
maps input "evan chang" into "evanchang@cs.colorado.edu". Note that we overloaded concat operator to allow for more than 2 operands.

### 4.2. Deductive Search Methodology

A simple search strategy is to enumerate all programs in order of increasing size [1] by doing a bottom-up enumeration of the grammar. This can be done by maintaining a graph of reachable values starting from the input state in the user-provided example. This simply requires access to the executable semantics of the operators in the DSL. Bottom-up enumeration is very effective for small grammar fragments since executing operators forward is very fast. Some techniques have been proposed to increase the scalability of enumerative search: (i) divide and conquer that decomposes the problem of finding programs that satisfy all examples to that of finding programs, each of which satisfies some subset, and then combining those programs using conditional predicates [2]. (ii) operator-specific lifting functions that can compute the output set from input sets more efficiently than point-wise computation. Lifting functions are essentially the forward transformer for an operator [30].

Unfortunately, bottom-up enumeration does not scale to large grammars because there are often too many constants to start out with. Our search methodology combines bottom-up enumeration with a novel top-down enumeration of the grammar. The top-down enumeration is goal-directed and requires pushing the specification across an operator using its inverse semantics. This is performed using *witness functions* that translate the specification for a program expression of the kind $F(e_1, e_2)$ to specifications for what the sub-expressions $e_1$ and $e_2$ should be. The bottom-up search first enumerates smaller sub-expressions before enumerating larger expressions. In contrast, the top-down search first fixes the top-part of an expression and then searches for its sub-expressions.

The overall top-down strategy is essentially a divide-and-conquer methodology that recursively reduces the problem of synthesizing a program expression $e$ of a certain kind and that satisfies a certain specification $\psi$ to simpler sub-problems (where the search is either over sub-expressions of $e$ or over sub-specifications of $\psi$), followed by appropriately combining those results. The reduction logic for reducing a synthesis problem to simpler synthesis problems depends on the nature of the involved expression $e$ and the inductive specification $\psi$. If $e$ is a non-terminal in the grammar, then the sub-problems correspond to exploring the various production rules corresponding to $e$. If $e$ is an operator application $F(e_1, e_2)$, then the sub-problems correspond to exploring multiple sub-goals for each parameter of that operator. As is usually the case with search algorithms, most of these explorations fail. PBE systems achieve real-time efficiency in practice by leveraging heuristics to predict which explorations are more likely to succeed and then either only explore those or explore them preferentially over others.

Machine learning techniques can be used to learn such heuristics in an effective manner. Below, we provide more details on one such method for a guided search in the deductive strategy [21].

### *4.3. ML-based Search Algorithm*

A key ingredient of the top-down search methodology mentioned above is grammar enumeration where while searching for a program expression $e$ of the non-terminal kind, we enumerate all the production rules corresponding to $e$ to obtain a new set of search problems and recursively solve each one of them. The goal of this work [21] was to determine the best production rules that we should explore while ignoring certain production rules that are unlikely to provide a desired program. Now, it might seem a bit outlandish to claim that we can determine the correct production rule to explore before even exploring it!

However, many times the provided input-output specification itself provides clues to make such a decision accurately. For example, in the context of the DSL mentioned in Figure 8, lets consider an example where the input is “evan” and the desired output is “evan@cs.colorado.edu”. In this case, even before exploring the productions rules, it is fairly clear that we should apply the concat operator instead of substr operator; a correct program is concat(Input, conststr(“@cs.colorado.edu”)). Similarly, if our input is “xinyu feng” and the desired output is “xinyu” then it is clear that we should apply the substr operator; a correct program is substr(Input, 1, pos(Input, Alphanumeric, “ ”, 1)).

But, exploiting the structure in input-output examples along with production rules is quite challenging as these are non-homogeneous structures without a natural vector space representation. Building upon recent advances in natural language processing, our ML based approach uses a version of neural networks to exploit the structure in input-output examples to estimate the set of best possible production rules to explore. For-

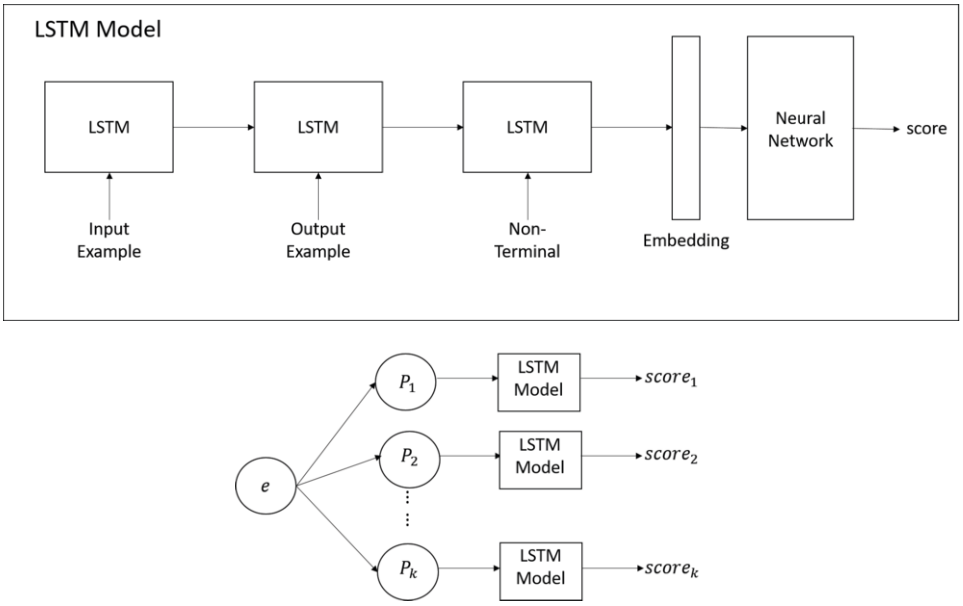


**Figure 9.** LSTM based model for computing score for the candidate set of production rules $P_1, \ldots, P_k$ during the grammar expansion process. The top figure shows details of the ML model used to compute score for a candidate production rule when placed in the context of the given input-output examples.

mally, given the input-output examples represented by $\psi$, and a set of candidate production rules $P_1, P_2, \ldots, P_k$ whose LHS is our current non-terminal $e$ we compute a score $s_i = score(\psi, P_i)$ for each candidate rule $P_i$. This score reflects the probability of synthesis of a desired program if we select rule $P_i$ for the given input-output examples $\psi$. Note that input-output example specification $\psi$ changes during the search process as we decompose the problem into smaller sub-problems; hence for recursive grammars, we need to compute the scores every time we wish to explore a production rule.

For learning the scoring model, similar to [6], our method embeds input-output examples in a vector space using a popular neural network technique called LSTM (Long Short-Term Memory) [17]. The embedding of a given input-output specification essentially captures its critical features, e.g., if input is a substring of output or if output is a substring of input etc. We then match this embedding against an embedding of the production rule $P_i$ to generate a joint embedding of $(\psi, P_i)$ pair. We then learn a neural network based function to map this joint embedding to the final score. Now for prediction, given scores $s_1, s_2, \ldots, s_k$, we select branches with top most scores with large enough margin, i.e., we select rules $P_{i_1}, \ldots, P_{i_\ell}$ for exploration where $s_{i_1} \geq s_{i_2} \cdots \geq s_{i_\ell}$ and $s_{i_\ell} - s_{i_{\ell+1}} \geq \tau$; $\tau > 0$ is a threshold parameter that we discuss later.

See Figure 9 for an overview of our LSTM based model and the entire pipeline.

To test our technique, we applied it to a much more expressive version of the Flash Fill DSL [10] that includes operators over richer data types such as numbers and dates. For training and testing our technique, we collected 375 benchmarks from real-world customer scenarios. Each benchmark consists of a set of input strings and their corresponding outputs. We selected 300 benchmarks for training and remaining 75 for testing.

| **Metric** | **PROSE** | **DC** | **RF** | **NGDS** |
|---|---|---|---|---|
| **Accuracy (% of 73)** | 67.12 | 32.88 | 16.44 | **68.49** |
| **Speed-up** ($\times$ **PROSE)** | 1.00 | 1.51 | 0.26 | **1.67** |

**Table 1.** (Table 1 of [21]) Accuracy and average speed-up of NGDS vs. baseline methods. Accuracies are computed on a test set of 73 tasks. *Speed-up* of a method is the geometric mean of it's per-task speed-up (ratio of synthesis time of PROSE and of the method) when restricted to a subset of tasks with PROSE's synthesis time is $\geq 0.5$ sec.

For each training benchmark, we generated top 1000 programs using existing top-down enumerative approach and logged relevant information for our grammar enumeration. For example, when we want to expand certain grammar symbol (say expr in Figure 8) with the goal of mapping given inputs to required outputs, we log all the relevant production rules $P_i$, $\forall i$ (i.e., rules in Line 1 of Figure 8). We also log the score $s_i$ of the top program that is generated by applying production rule $P_i$. That is, each training instance is $(\psi, P_i, s_i)$ for a given node with input-output examples $\psi$. We use standard DNN tools to train the model for grammar enumeration. That is, whenever we need to decide on which production rule to select for expansion, we compute score for each possible rule $P_i$ and select the rules whose scores are higher than the remaining rules by a margin of $\tau$.

Threshold $\tau$ is an interesting knob that helps decide between exploration vs exploitation. That is, smaller $\tau$ implies that we trust our ML model completely and select the best choice presented by the model. On the other hand, larger $\tau$ forces system to be more conservative and use ML model sparingly when it is highly confident. For example, on the 75 test benchmarks, setting $\tau = 0$ i.e. selecting ML model's predicted production rule for every grammar expansion decision, we select the best production rule 92% of the instances. Unfortunately, selecting wrong production rule 8% of the times might lead to synthesis of a relatively poor program or in worst case, no program. However, by increasing $\tau = 0.1$ we can increase our chances of selection of the best production rule to 99%. Although in this case, for nearly 50% instances the ML model does not differentiate between production rules, i.e., the predicted scores are all within $\tau = 0.1$ length interval. Hence, we enumerate all the rules in about 50% of the grammar expansion instances and are able to prune production rules in only 50% cases. Nonetheless, this itself leads to impressive computation time improvement of up to 12x over naïve exploration for many challenging test benchmarks. Table 1 presents average speed-up obtained by our method (NGDS) over the naïve exploration technique used by the PROSE system as well as two of the existing deep learning based techniques: RobustFill [6] and DeepCoder [3]. RobustFill (RF) does not leverage the deductive search structure and instead tries to synthesize programs end-to-end using deep learning. As seen by the table, the accuracy of such a system is not very good and in fact, even the overall computation cost is also significantly worse than NGDS. DeepCoder (DC) is a technique that imposes a static priority over operators to be explored during deductive search. So unlike NGDS, DeepCoder does not change the priority list over operators with each step's input-output pair.

## 5. Ranking

Examples are a severe under-specification of the user's intent in many useful task domains. As a result, several programs in an underlying DSL are consistent with a given set of training examples, but are unintended, i.e., they would produce an undesired output on some test inputs. Usability concerns further necessitate that we learn an intended program from as few examples as possible.

PBE systems address this challenge by leveraging a ranking scheme to select between different programs consistent with the examples provided by the user. Ideally, we want to bias the ranking of programs so that *natural* programs are ranked higher. While the notion of *naturalness* of programs is highly subjective, still in practice, one can see certain succinct patterns associated with natural programs that one can try to capture via real-world training datasets.

The ranking can either be performed in a phase subsequent to the one that identifies the many programs that are consistent with the examples [38], or it can be in-built as part of the search process [25,3]. Furthermore, the ranking can be a function of the program structure or additional test inputs.

### *5.1. Ranking based on Program Structure*

A basic ranking scheme can be specified by defining a preference order over program expressions based on their features. Two general principles that are useful across various domains are: prefer small expressions (inspired by the classic notion of Kolmogorov complexity) and prefer expressions with fewer constants (to force generalization). For specific DSLs, more specific preferences or features can be defined based on the operators that occur in the DSL.

### *5.2. Ranking based on test inputs*

The likelihood of a program being the intended one not only depends on the structure of that program, but also on features of the input data on which that program will be executed and the output data produced by executing that program. In some PBE settings, the synthesizer often has access to some additional test inputs on which the intended program is supposed to be executed. Singh showed how to leverage these additional test inputs to guess a reduction in the search space with the goal to speed up synthesis and rank programs better [35]. Ellis and Gulwani observed that the additional test inputs can be used to re-rank programs based on how similar are the outputs produced by those programs on the test inputs to the outputs in the training/example inputs provided by the user [7].

For instance, consider the task of extracting years from input strings of the kind shown in the table below.

| Input | Output |
|---|---|
| Missing page numbers, 1993 | 1993 |
| 64-67, 1995 | 1995 |

The program $P1$: “Extract the last number” can perform the intended task. However, if the user provides only the first example, another reasonable program that can be synthesized is $P2$: “Extract the first number”. There is no clear way to rank $P1$ higher than $P2$ from just examining their structure. The above However, the output produced by $P1$ (on the various test inputs), namely $\{1993, 1995, \ldots, \}$ is a more meaningful set (of 4 digit numbers that are likely years) than the one produced by $P2$, namely (which manifests greater variability). The meaningfulness or similarity of the generated output can be captured via various features such as IsYear, numeric deviation, IsPersonName, and number of characters.

## 5.3. ML-based Ranking Function

Typically, *natural* or intended programs tend to have subtle properties that cannot be captured by just one feature or by an arbitrary combination of the multiple features identified above; empirical results presented in Table 2 confirms this hypothesis where the accuracy of the shortest program based ranker or a random ranker is poor. Hence, we need to learn a ranking function that appropriately combines the features in order to produce the intended natural programs. In fact, learning rankers over programs/sub-expressions represents an exciting domain where insights from ML and PL can have an interesting and impactful interplay.

Below, we present one such case study where we learn a ranking function that ranks sub-expressions and programs during the search process itself [26]. We learn the ranking function using training data that is extracted from diverse real-world customer scenarios. However learning such a ranking function that can be used to rank sub-expressions during the search process itself poses certain unique challenges. For example, we need to rank various non-homogeneous sub-expressions during each step of the search process but the feedback about our ranking decisions is provided only after synthesis of the final program. Moreover, the ranking function captures the intended program only if the final program is correct, hence, a series of "correct" ranking decisions over various sub-expressions might be nullified by one incorrect ranking decision.

To solve the above set of problems, we implement a simple program embedding based approach. Consider a program $P$ whose AST is given by $\mathscr{A}(P)$. Then the embedding of $P$ is computed recursively where $\phi(\mathscr{A}(P)) = \sum_i w_i \phi(\mathscr{A}(P_i))$, $P_i$ are the children of $P$ in $\mathscr{A}(P)$. Now leaf nodes of $\mathscr{A}(P)$ are embedded in $d$-dimensions using a few operator-specific features. We now pose the ranking problem as: find $\theta \in \mathbb{R}^d$ s.t. $\sum_j \theta_j \phi(P_a)_j \geq \sum_j \theta_j \phi(P_b)_j$ where $P_a$ is a "correct" program, i.e., it produces desired output on training datasets and $P_b$ is an "incorrect" program. $\theta_j$ and $\phi(P)_j$ represents the $j^{th}$ coordinate of $\theta$ and $\phi(P)$ respectively.

Now recall that our goal is to ensure that all the ranking decisions in benchmark are correct, so we need to use a different metric than the standard classification metric (see [26] for more details).

For learning $\theta$ as well as weights $w_i$, we use training benchmarks where each benchmark consists of a set of inputs and their corresponding outputs. For each benchmark, we synthesize 1000 programs using the first input-output pair in that benchmark, treating it as an example input-output pair. We categorize a synthesized program as "correct" if it generates correct output on all the other benchmark inputs, and "incorrect" otherwise. We then embed each sub-expression and the program in $d$-dimensional space using hand-crafted features. Our features reflect certain key properties of the programs, e.g., length of the program etc. We then use straightforward block-coordinate descent based methods to learn $\theta$, $w_i$'s in an iterative fashion.

*Empirical Results*: similar to search experiments, we learn our ranking function using a collection of important benchmarks from real-world customer scenarios. We select about 100 benchmarks for training and test our system on the remaining 640 bench-

| Ranking Method | Acc@1 | | Acc@10 | |
|---|---|---|---|---|
| | $m=1$ | $m=2$ | $m=1$ | $m=2$ |
| RANDOM | 0.22 | 0.60 | 0.38 | 0.67 |
| (A) SHORTEST PROGRAM | 0.37 | 0.69 | 0.49 | 0.80 |
| (B) FEWER CONSTANTS | 0.38 | 0.60 | 0.59 | 0.80 |
| (A) and (B) | 0.44 | 0.72 | 0.60 | 0.87 |
| ML-based Ranker | 0.65 | 0.81 | **0.79** | **0.92** |

**Table 2.** Ranking: table compares precision@1 and precision@10 accuracy for various methods when supplied different number of input-output example pairs ($m = 1,2$). Our ML-ranker provides significantly higher accuracy and estimates correct program for 65% test benchmarks using just one input-output example.

marks. We evaluate performance of our ranker using precision k metric. That is, precision k is the fraction of test benchmarks in which at least one "correct" program lies in the top-$k$ programs (as ranked by our ranker). We also compute precision k for different specification sizes, i.e., for different number of input-output examples being supplied.

Table 2 compares accuracy (measured in precision@k) of our method with four baselines: a) random ranker that at each node selects a random sub-expression, b) shortest program which selects programs with the smallest number of operators. c) program that selects the smallest number of constants. d) a linear combination of the shortest and smallest constants heuristics. Note that with 1 input-output example, our method is almost 50% more accurate than baselines. Naturally with 2 examples, baselines' performance also improves as there fewer programs that satisfy 2 examples.

Additionally, we can learn individual $\theta$ for each user/organization thus leading to personalized ranker. For example, our method can learn processing an input string as "European" style date-time instead of "American" style date-time.

## 6. Interactivity

While use of ranking in the synthesis methodology attempts to avoid selecting an unintended program, it cannot always succeed. Hence, it is important to design appropriate user interaction models for the PBE paradigm that can provide the equivalent of debugging experience in standard programming environments. There are two important goals for a user interaction model that is associated with a PBE technology [24]. First, it should provide transparency to the user about the synthesized program(s). Second, it should guide the user in resolving ambiguities in the provided specification.

In order to facilitate transparency, the synthesized program can be displayed to the user. In that context, it would be useful to have readability as an additional criterion during synthesis. The program can also be paraphrased in natural language, especially to facilitate understanding by non-programmers.

In order to resolve ambiguities, we can present multiple synthesized programs to the user and ask the user to pick between those. More interestingly, we can also leverage availability of other test input data on which the synthesized program is expected to be executed. This can be done in few different ways. A set of representative test inputs can be obtained by clustering the test inputs and picking a representative element from each cluster [28]. The user can then check the results of the synthesized program on those representative inputs. Alternatively, clustering can also be performed on the outputs produced by the synthesized program. Yet, another approach can be to leverage *distinguishing inputs* [19]. The idea here is to synthesize multiple programs that are consistent with the examples provided by the user but differ on some test inputs. The PBE system can then ask the user to provide the intended output on one or more of these distinguishing inputs. The choice for the distinguishing input to be presented to the user can be based on its expected potential to distinguish between most of those synthesized programs.

There are many heuristic decisions in the above-mentioned interaction models that can ideally be learned using ML techniques such as what makes a program more readable, or which set of programs to present to the user, or how to cluster the input or output column. Below, we discuss one such investigation related to clustering of strings.

### *6.1. Clustering of Strings*

We propose an agglomerative hierarchical clustering based method for clustering the strings. Intuitively, we want to cluster strings together which can be represented by a specific but natural regular expression. For example, given strings $\{1990, 1995, 210\text{BC}, 450\text{BC}\}$, we want to find the two clusters represented by regular expressions $\text{Digit}^4$ and $\text{Digit}^3 \cdot \text{BC}$.

We find the tightest and natural regular expression representing a given set of strings using program synthesis over a regular expression specific language. [28]. Our algorithm randomly samples a few strings and then finds the most likely regular expressions by synthesizing them using pairs of strings. The most highly rates regular expressions can be thought of as cluster representatives. We then define a distance function that computes distance of a string to a regular expression. Using this distance function, we then apply standard agglomerative hierarchical clustering algorithm to obtain representative regular expressions.

For example, given strings from a dataset containing postal codes such as: {99518, 61021-9150, 2645, K0K 2C0, 61604-5004...}, our system finds clusters such as:

- $\text{Digit}^5$
- $\text{Digit}^4$
- $\text{UpperCase} \cdot \text{Digit} \cdot \text{UpperCase Digit} \cdot \text{UpperCase} \cdot \text{Digit}$
- $61\text{Digit}^3 - \text{Digit}^4$
- S7K7K9

Note that the regular expressions are able to capture the key clusters such as $\text{Digit}^5$ etc, but it also captures certain anomalies such as S7K7K9. We also evaluate our system over real-world datasets using Normalized Mutual Information (NMI) metric which is a standard clustering metric. We observe that if given enough computation time, our system is able to obtain nearly optimal NMI of $\approx 1.0$. Moreover, by appropriately sampling and synthesizing regular expressions, we can speed up the computation by a factor of 2 despite recovering clusters with NMI of 0.95. We refer the interested readers to [28] for more details.

## 7. Future Directions

*Applications* Robotic Process Automation (RPA) can be another killer application for program synthesis. The goal in RPA is to automate high-volume rules-driven business processes that often connect different applications. These typically require logging into IT systems and copying and pasting data across systems. For instance, consider the task of opening an invoice in a PDF format that is received as an attachment in an email, extracting various fields from the invoice, and entering them inside multiple systems. Program synthesis technologies can help synthesize such scripts from few demonstrations by the business user.

Another interesting application of program synthesis can be in the space of programming real-world robots. General-purpose programmable robots may be a common household entity in a few decades from now. Each household will have its own unique geography for the robot to navigate and a unique set of chores for the robot to perform. Example-based training could be an effective means for programming robots for personalized tasks and personalized household environments.

*Performant Synthesis* The synthesized scripts might need to be executed on big data. In such a scenario, it is desirable to synthesize not just a correct program that meets the intent, but one that is also efficient and hence does not waste computational resources.

Often there are many different programs to accomplish a particular task. These programs may not be semantically equivalent but they have the same behavior on the kinds of inputs they are expected to be executed on. The ranking schemes in program synthesis are generally tuned to pick any of these *correct* programs. However, some of these programs may be much more efficient than the other, and it may be desirable to pick one such efficient program. For instance, suppose the goal is to extract LastName from inputs of kind “FirstName LastName”. One correct program to accomplish such a task can operate by extracting the second word, while another correct program to accomplish the same task can operate by extracting all characters after the last space. It turns out that the latter program is much more efficient than the former since it avoids use of regular expressions.

*Readable Synthesis* The synthesized scripts might need to be readable and modifiable. In some scenarios, it is not important to inspect the code of the synthesized program, especially when the goal is to execute the script for a one-off task and wherein the correctness of the underlying transformation can be verified by visual inspection over small input data. Applying Flash Fill [10] on small-sized input columns is an example of such a scenario, wherein an end user may simply inspect the derived column to verify that the string transformation has been performed correctly. However, if the input on which the synthesized script is to be executed is large, or if the synthesized script needs to be executed multiple times in the future, then the user may want to inspect, and possibly even edit the code of the underlying synthesized program. In such scenarios, it is important to synthesize code that is readable. Furthermore, such a code may need to be synthesized in a specific target language desired by the user. This leads to many interesting research challenges such as leveraging idiomatic patterns and libraries that the user is familiar with, choice of variable names, and formatting of code. Another interesting concern relates to maintainability of such synthesized code. For instance, if the user provides additional examples in the future to adapt the behavior of the code on new additional inputs, then what happens to any changes that the user may have made in the old synthesized code? One interesting possibility is to ensure that the newly synthesized code is as similar to the old synthesized code as possible, which can be regarded as automation of test-driven development [29].

Synthesizing readable code in specific target languages shall allow PBE technologies to be integrated inside main-stream coding workflows such as IDEs or notebooks.

*Multi-model intent specification* While this article has focused on leveraging examples as specification of intent, certain class of tasks are best described using natural language such as spreadsheet queries [14] and smartphone scripts [23]. The next generation of programming experience shall be built around multi-modal specifications that are natural and easy for the user to provide. The new paradigm shall allow expressing intent using combination of various means [32] such as examples, demonstrations, natural language, keywords, and sketches [42].

*Predictive Synthesis* For some task domains, it is often possible to predict the user's intent without any input-output examples, i.e., from input-only examples. For instance, extracting tables from web pages, PDF documents, or log files, or splitting a column into multiple columns [30]. While providing examples is already much more convenient than authoring one-off scripts, there are scenarios where providing examples can be quite tedious overall. For instance, consider the task of extracting fields from a log file. If the number of fields is large, then providing examples for each field would be quite tedious. Having the system guess the user's intent without any examples can also power novel user experiences such as enabling question-answering on semi-structured data, wherein the system can automatically infer the underlying relational tabular structure without requiring the user to provide any examples.

*Adaptive Synthesis* Another interesting future direction is to build systems that learn user preferences based on past user interactions across different programming sessions. For instance, the underlying ranking can be dynamically updated. This can pave the way for personalization of PBE technologies to specific users, as well as enable learning across users in a given organization or cloud. Tasks that required more examples earlier can now be accomplished with fewer examples. In fact, this can also facilitate predictive synthesis. For instance, consider the task of parsing a custom log file or extracting a table from a web page. Initially, a user may have to provide some examples. In the future, when the same user or even a different user, is faced with the same task but on a different input of the same format, the underlying adaptive synthesis system should be able to handle the task predictively, i.e., without any examples.

*PL meets ML* While PL has democratized access to machine implementations of precise ideas, ML has democratized access to discovering heuristics to deal with fuzzy and noisy situations. The new AI revolution requires frameworks that can facilitate creation of AI-infused software and applications. Synergies between PL and ML can help lay the foundation for construction of such frameworks [34,5,8,41].

For instance, language features can be developed that allow the developer to express non-determinism with some default resolution strategies that can then automatically get smarter with usage. As opposed to traditional AI based domains such as vision, text, bioinformation, such self-improving systems present entirely different data formats and pose unique challenges that foreshadow an interesting full-fledged research area with opportunity to impact how we program and think about interacting with computer systems in general.

## 8. Conclusion

PBE is a new frontier in AI and is set to revolutionize the programming experience. The technology has already matured to the extent that it can provide 10-100x productivity increase in many task domains for both data scientists and developers. The two killer applications for PBE today are: data wrangling and code refactoring. Data scientists spend 80% time wrangling data while developers spend up to 40% time refactoring code in a typical application migration scenario. Another significant aspect of PBE is its potential to enable programming for the masses, given that 99% people who use computers do not know programming.

We have leveraged inspiration from both logical reasoning and machine learning to build usable and practical PBE systems. The Microsoft PROSE SDK [1] exposes generic search and ranking algorithms, allowing advanced developers to construct PBE capabilities for new task domains. This SDK has been used to build product-quality implementations of many PBE capabilities that have shipped through multiple Microsoft products across Office, Windows, SQL, and Azure.

A key challenge in PBE is to search for programs that are consistent with the examples provided by the user. On the symbolic reasoning side, our search methodology in PBE leverages two key ideas: restrict the search to a domain-specific programming language (PL) specified as a grammar, and perform a goal-directed top-down search that leverages inverse semantics of operators to decompose a goal into a choice of multiple sub-goals. However, this search can be made even more tractable by learning tactics (using ML) to prefer certain choices over others during both grammar exploration and sub-goal selection.

Another key challenge in PBE is to understand the user's intent in the face of ambiguity that is inherent in example-based specifications, and furthermore, to understand it from as few examples as possible. For this, we leverage use of a ranking function with the goal of the search now being to pick the highest ranked program that is consistent with the examples provided by the user. The ranking is a function of various symbolic features of a program such as size, number of constants, use of a certain combination of operators. The ranking is also a function of the outputs generated by the program (non-null or not, same type as the example outputs or not) and more generally the execution traces of the program on new test inputs. While various PL concepts go into defining the features of a ranking function, ML-based techniques can be used to build models over these different classes of features.

A third challenge relates to debuggability: provide transparency to the user about the synthesized program and help the user to refine the specification in an interactive loop. We have investigated user interaction models that leverage concepts from both PL and ML including active learning based on synthesis of multiple top-ranked programs (each of which is consistent with the user's specification) and leveraging their differences, clustering of inputs to identify various input classes and hence representative inputs, clustering of outputs to identify any potential discrepancies, and navigation through a large program set represented succinctly as a grammar.

The above-mentioned directions highlight opportunities to design novel techniques that combine logical reasoning based symbolic methods developed in the PL community

[1] https://microsoft.github.io/prose/

with ML methods to solve various challenges that arise in construction of efficient, robust, and usable PBE systems. We believe that the ongoing AI revolution shall further drive novel synergies between PL and ML to facilitate creation of intelligent software in general. PBE systems, and more generally program synthesis systems, that relate to real-time intent understanding are a great case study for investigating ideas in this space.

Programming has evolved from use of punched cards and low-level assembly language programming to programming with high-level languages in beautiful code editors. The next evolution will leverage advances in program synthesis techniques to take programming closer to natural human communication, wherein it will become multi-modal and will involve use of various forms of intent expression including examples and natural language. Today, examples are already present in programming in the form of test cases, and comments are nothing but natural-language-based specifications. However, these artifacts, namely test cases and comments, are today constructed after code has been written in order to test code or to document code. The next frontier will lift these artifacts to first-class citizens for the process of authoring code itself.

## References

[1] R. Alur, R. Bodík, E. Dallal, D. Fisman, P. Garg, G. Juniwal, H. Kress-Gazit, P. Madhusudan, M. M. K. Martin, M. Raghothaman, S. Saha, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *Dependable Software Systems Engineering*, pages 1–25. 2015.

[2] R. Alur, A. Radhakrishna, and A. Udupa. Scaling enumerative program synthesis via divide and conquer. In *TACAS*, pages 319–336, 2017.

[3] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow. Deepcoder: Learning to write programs. In *ICLR*, 2017.

[4] D. W. Barowy, S. Gulwani, T. Hart, and B. G. Zorn. FlashRelate: extracting relational data from semi-structured spreadsheets using examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 218–228, 2015.

[5] P. Bielik, V. Raychev, and M. T. Vechev. Programming with ”big code”: Lessons, techniques and applications. In *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA*, pages 41–50, 2015.

[6] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A. Mohamed, and P. Kohli. Robustfill: Neural program learning under noisy I/O. In *ICML*, 2017.

[7] K. Ellis and S. Gulwani. Learning to learn programs from examples: Going beyond program structure. In *IJCAI*, pages 1638–1645, 2017.

[8] J. K. Feser, M. Brockschmidt, A. L. Gaunt, and D. Tarlow. Neural functional programming. *CoRR*, abs/1611.01988, 2016.

[9] A. Graves, G. Wayne, M. Reynolds, T. Harley, I. Danihelka, A. Grabska-Barwinska, S. G. Colmenarejo, E. Grefenstette, T. Ramalho, J. Agapiou, A. P. Badia, K. M. Hermann, Y. Zwols, G. Ostrovski, A. Cain, H. King, C. Summerfield, P. Blunsom, K. Kavukcuoglu, and D. Hassabis. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, 2016.

[10] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, pages 317–330, 2011.

[11] S. Gulwani. Programming by examples - and its applications in data wrangling. In *Dependable Software Systems Engineering*, pages 137–158. 2016.

[12] S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, 2012.

[13] S. Gulwani and P. Jain. Programming by examples: PL meets ML. In *Programming Languages and Systems - 15th Asian Symposium APLAS, Suzhou, China*, volume 10695 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 2017.

[14] S. Gulwani and M. Marron. Nlyze: interactive programming by natural language for spreadsheet data analysis and manipulation. In *SIGMOD*, pages 803–814, 2014.

[15] S. Gulwani, O. Polozov, and R. Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017.

[16] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *PLDI*, pages 317–328, 2011.

[17] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, Nov. 1997.

[18] S. Itzhaky, S. Gulwani, N. Immerman, and M. Sagiv. A simple inductive synthesis methodology and its applications. In *OOPSLA*, pages 36–46, 2010.

[19] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, pages 215–224, 2010.

[20] A. Joulin and T. Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. In *NIPS*, pages 190–198, 2015.

[21] A. Kalyan, A. Mohta, O. Polozov, D. Batra, P. Jain, and S. Gulwani. Neural-guided deductive search for real-time program synthesis from examples. In *International Conference on Learning Representations*, 2018.

[22] V. Le and S. Gulwani. FlashExtract: a framework for data extraction by examples. In *PLDI*, pages 542–553, 2014.

[23] V. Le, S. Gulwani, and Z. Su. Smartsynth: synthesizing smartphone automation scripts from natural language. In *The 11th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 193–206, 2013.

[24] M. Mayer, G. Soares, M. Grechkin, V. Le, M. Marron, O. Polozov, R. Singh, B. G. Zorn, and S. Gulwani. User interaction models for disambiguation in programming by example. In *UIST*, pages 291–301, 2015.

[25] A. K. Menon, O. Tamuz, S. Gulwani, B. W. Lampson, and A. Kalai. A machine learning framework for programming by example. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, pages 187–195, 2013.

[26] N. Natarajan, N. Datha, D. Simmons, S. Gulwani, and P. Jain. Learning natural programs from a few examples in real-time. In *AIStats*, 2019.

[27] A. Neelakantan, Q. V. Le, and I. Sutskever. Neural programmer: Inducing latent programs with gradient descent. *CoRR*, abs/1511.04834, 2015.

[28] S. Padhi, P. Jain, D. Perelman, O. Polozov, S. Gulwani, and T. D. Millstein. Flashprofile: a framework for synthesizing data profiles. *PACMPL*, 2(OOPSLA):150:1–150:28, 2018.

[29] D. Perelman, S. Gulwani, D. Grossman, and P. Provost. Test-driven synthesis. In *PLDI*, pages 408–418, 2014.

[30] M. Raza and S. Gulwani. Automated data extraction using predictive program synthesis. In *AAAI*, pages 882–890, 2017.

[31] M. Raza, S. Gulwani, and N. Milic-Frayling. Programming by example using least general generalizations. In *AAAI*, pages 283–290, 2014.

[32] M. Raza, S. Gulwani, and N. Milic-Frayling. Compositional program synthesis from natural language and examples. In *IJCAI*, pages 792–800, 2015.

[33] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 404–415, 2017.

[34] C. Simpkins. Integrating reinforcement learning into a programming language. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*, 2010.

[35] R. Singh. Blinkfill: Semi-supervised programming by example for syntactic string transformations. *PVLDB*, 9(10):816–827, 2016.

[36] R. Singh and S. Gulwani. Learning semantic string transformations from examples. *PVLDB*, 5(8):740–751, 2012.

[37] R. Singh and S. Gulwani. Synthesizing number transformations from input-output examples. In *CAV*, pages 634–651, 2012.

[38] R. Singh and S. Gulwani. Predicting a correct program in programming by example. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 398–414, 2015.

[39] R. Singh and S. Gulwani. Transforming spreadsheet data types using examples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 343–356, 2016.
[40] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *PLDI*, pages 15–26, 2013.
[41] R. Singh and P. Kohli. AP: artificial programming. In *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA*, pages 16:1–16:12, 2017.
[42] A. Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, University of California, Berkeley, 2008.
[43] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *POPL*, pages 313–326, 2010.

*Engineering Secure and Dependable Software Systems*
*A. Pretschner et al. (Eds.)*
*IOS Press, 2019*
*© 2019 The authors and IOS Press. All rights reserved.*
*doi:10.3233/978-1-61499-977-5-83*

# Automatic Program Verification with SEAHORN

Arie GURFINKEL [a,1] and Jorge A. NAVAS [b]
[a] *Department of Electrical and Computer Engineering, University of Waterloo*
[b] *SRI International*

**Abstract.** In this paper, we present SEAHORN, a software verification framework. The key distinguishing feature of SEAHORN is its modular design that separates the concerns of the syntax of the programming language, its operational semantics, and the verification semantics. SEAHORN encompasses several novelties: it (a) encodes verification conditions using an efficient yet precise inter-procedural technique, (b) provides flexibility in the verification semantics to allow different levels of abstraction, (c) uses Horn-clauses as an intermediate language to represent verification conditions which simplifies interfacing with multiple verification tools based on Horn-clauses, and (d) leverages the state-of-the-art in software model checking and abstract interpretation for verification. SEAHORN provides users with a powerful verification tool and researchers with an extensible and customizable framework for experimenting with new software verification techniques.

**Keywords.** software Model Checking, program verification, Constrained Horn Clauses, Abstract Interpretation

## 1. Introduction

In this paper, we describe the SEAHORN verification framework. SEAHORN extends the LLVM [78] compiler infrastructure with verification techniques based on Software Model Checking and Abstract Interpretation. The framework provides many components that can be combined together for a variety of analysis needs. Many useful analyzers (e.g., memory safety, overflow checker, null pointer checker, etc.) are provided out of the box. The paper presents an overview of the framework and detailed description of the two verification engines: SPACER for Model Checking and CRAB for Abstract Interpretation.

In the rest of this section, we summarize the key unique features of the framework. First, SEAHORN decouples a programming language syntax and semantics from the underlying verification technique. Different programming languages include a diverse assortments of features, many of which are purely syntactic. Handling them fully is a major effort for new tool developers. We tackle this problem in SEAHORN by separating the language syntax, its operational semantics, and the underlying verification semantics – the semantics used by the verification engine. Specifically, we use the LLVM front-end(s) to deal with the idiosyncrasies of the syntax. We use LLVM intermediate repre-

[1]Corresponding Author: Arie Gurfinkel, Department of Electrical and Computer Engineering, University of Waterloo, Canada, E-mail: arie.gurfinkel@uwaterloo.ca.

sentation (IR), called the *bitcode*, to deal with the operational semantics, and apply a variety of transformations to simplify it further. In principle, since the bitcode has been formalized [102], this provides us with a well-defined formal semantics. Finally, we use Constrained Horn Clauses (CHC) to logically represent the verification conditions (VC).

Second, SEAHORN provides an efficient and precise analysis of programs with procedures using inter-procedural (i.e., modular) verification techniques. SEAHORN summarizes the input-output behavior of procedures efficiently without inlining. The expressiveness of the summaries is not limited to linear arithmetic, but extends to richer logics, including, for instance, arrays. Furthermore, SEAHORN includes a program transformation that lifts deep assertions closer to the main procedure. This increases context-sensitivity of intra-procedural analyses (used both in verification and compiler optimization), and has a significant impact on our inter-procedural verification algorithms.

Third, SEAHORN allows developers to customize the verification semantics and offers users with verification semantics of various degrees of abstraction. SEAHORN is fully parametric in the (small-step operational) semantics used for the generation of VCs. The level of abstraction in the built-in semantics varies from considering only LLVM numeric registers to considering the whole heap (modeled as a collection of non-overlapping arrays). In addition to generating VCs based on small-step semantics [90], SEAHORN can also automatically lift small-step semantics to large-step [6, 57] (a.k.a. Large Block Encoding, or LBE).

Fourth, SEAHORN uses Constrained Horn Clauses (CHC) as its intermediate verification language. CHC provide a convenient and elegant way to formally represent many encoding styles of verification conditions. The recent popularity of CHC as an intermediate language for verification engines makes it possible to interface SEAHORN with a variety of new and emerging tools.

Fifth, SEAHORN builds on the state-of-the-art in Software Model Checking (SMC) and Abstract Interpretation (AI). SMC and AI have independently led over the years to the production of analysis tools that have a substantial impact on the development of real world software. Interestingly, the two exhibit complementary strengths and weaknesses (see e.g., [1, 9, 46, 56]). While SMC so far has been proved stronger on software that is mostly control driven, AI is quite effective on data-dependent programs. SEAHORN combines SMT-based model checking techniques with program invariants supplied by an Abstract Interpreter.

Finally, SEAHORN is open-sourced and is implemented on top of the open-source LLVM compiler infrastructure. LLVM is a well-maintained, well-documented, and continuously improving framework. This allows SEAHORN users to easily integrate program analyses, transformations, and other tools that targets LLVM. Moreover, since SEAHORN analyses LLVM IR, this allows to exploit a rapidly-growing frontier of LLVM front-ends, encompassing a diverse set of languages. SEAHORN itself is released as open-source as well (source code can be downloaded from `http://seahorn.github.io`).

The design of SEAHORN provides users, developers, and researchers with an extensible and customizable environment for experimenting with and implementing new software verification techniques. SEAHORN is implemented in C++ in the LLVM compiler infrastructure [78]. The overall approach is illustrated in Figure 1. SEAHORN has been developed in a modular fashion; its architecture is layered in three parts:

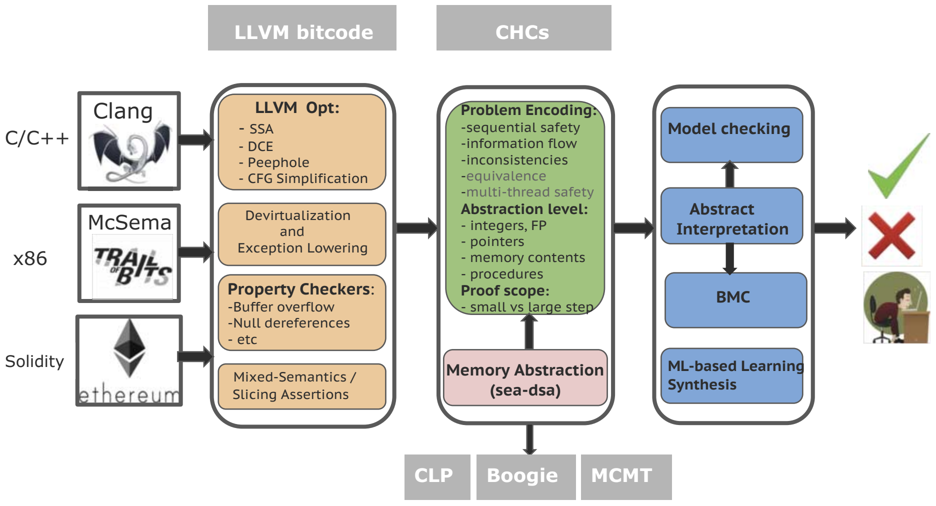


**Figure 1.** Overview of SEAHORN architecture.

**Front-End:** Takes an LLVM based program (e.g., C) input program and generates LLVM IR bitcode. Specifically, it performs the pre-processing and optimization of the bitcode for verification purposes. More details are reported in Section 2.

**Middle-End:** Takes as input the optimized LLVM bitcode and emits verification condition as Constrained Horn Clauses (CHC). The middle-end is in charge of selecting the encoding of the VCs and the degree of abstraction. VCs can be exported to different formats such as Constraint Logic Programming (CLP), Boogie or MCMT (Model Checking Modulo Theories). More details are reported in Section 3.

**Back-End:** Takes CHC as input and outputs the result of the analysis. In principle, any verification engine that digests CHC clauses could be used to discharge the VCs. Currently, SEAHORN employs an SMT-based model checking engine SPACER [74]. Complementary, SEAHORN uses the abstract interpretation-based analyzer CRAB for providing numerical invariants. More details are reported in Section 4.

*Related Work.* Automated analysis of software is an active area of research. There is a large number of tools with different capabilities and trade-offs [5, 7, 8, 18, 20–22, 32, 82]. Our approach on separating the program semantics from the verification engine has been previously proposed in numerous tools. From those, the tool SMACK [91] is the closest to SEAHORN. Like SEAHORN, SMACK targets programs at the LLVM-IR level. However, SMACK targets Boogie intermediate verification language [35] and Boogie-based verifiers to construct and discharge the proof obligations. SEAHORN differs from SMACK in several ways: (i) SEAHORN uses CHC as its intermediate verification language, which allows to target different solvers and verification techniques (ii) it tightly integrates and combines both state-of-the-art software model checking techniques and abstract interpretation and (iii) it provides an automatic inter-procedural analysis to reason modularly about programs with procedures.

Inter-procedural and modular analysis is critical for scaling verification tools and has been addressed by many researchers (e.g., [2, 67, 74, 77, 80, 96]). Our approach of using mixed-semantics [63] as a source-to-source transformation has been also explored in [77]. While in [77], the mixed-semantics is done at the verification semantics (Boogie in this case), in SEAHORN it is done in the front-end level allowing mixed-semantics to interact with compiler optimizations.

Constrained Horn clauses have been proposed [12] as an intermediate (or exchange) format for representing verification conditions. However, they have long been used in the context of static analysis of imperative and object-oriented languages (e.g., [81, 90]) and more recently adopted by an increasing number of solvers (e.g., [13, 42, 67, 74, 80]) as well as other verifiers such as UFO [3], HSF [53], VeriMAP [33], Eldarica [93], and TRACER [68].

A previous version of this paper has appeared in [58].

## 2. Pre-processing for Verification

In our experience, performance of even the most advanced verification algorithms is significantly impacted by the front-end transformations. In SEAHORN, the front-end plays a very significant role in the overall architecture.

In principle, SEAHORN can take any input program that can be translated into LLVM bitcode. However, SEAHORN has been highly customized to analyze C programs as translated by `clang`, and thus, analysis of C code is the most prominent SEAHORN's strength. More recently, SEAHORN has increasingly supported analysis of C++ programs, and we plan to continue doing that.

Our goal is to make SEAHORN not to be limited to C or C++ programs, but applicable (with various degrees of success) to a broader set of languages based on LLVM (e.g., Objective C, Rust, and Swift). For instance, we have recently added support for two very different languages: x86 binary programs using the McSema [98] tool, and Solidity smart contracts using a just-in-time compiler for Ethereum EVM code [37]. Although the verification results have been relatively modest compared with verification of C programs, they demonstrate the broad applicability of SEAHORN.

Once we have obtained LLVM bitcode, the front-end is split into two main subcomponents. The first one is a pre-processor that performs optimizations and transformations. This pre-processing is largely optional. Its main goal is to transform the LLVM bitcode to make the verification task *easier*. The second part is focused on a reduced set of transformations mostly required to produce correct results even if the pre-processor is disabled. It also performs SSA transformation and internalizes functions, but in addition, lowers `switch` instructions into `if-then-else`s, ensures only one exit block per function, inlines global initializers into the main procedure, and identifies `assert`-like functions.

The front-end can optionally inline functions. This is often useful to simplify verification tasks, and is also necessary for precise Bounded Model Checking (and, currently, is required for counterexample generation).

One typical problem in proving safety of large programs is that assertions can be nested very deep inside the call graph. As a result, counterexamples are longer and it is harder to decide for the verification engine what is relevant for the property of interest.

| | | | |
|---|---|---|---|
| *main* () | $main_{new}$ () | $p1_{entry}$ : | $p1_{new}$ () |
|     $p1$ (); $p1$ (); |     if (*) goto $p1_{entry}$; |     if (*) goto $p2_{entry}$; |     $p2_{new}$ (); |
|     assert ($c1$); |     else $p1_{new}$ (); |     else $p2_{new}$ (); |     assume ($c2$); |
| $p1$ () |     if (*) goto $p1_{entry}$; |     if ($\neg c2$) goto *error*; | $p2_{new}$ () |
|     $p2$ (); |     else $p1_{new}$ (); | $p2_{entry}$ : |     assume ($c3$); |
|     assert ($c2$); |     if ($\neg c1$) goto *error*; |     if ($\neg c3$) goto *error*; | |
| $p2$ () |     assume (false); |     assume (false); | |
|     assert ($c3$); | | *error* : assert (false); | |

**Figure 2.** A program before (left) and after (right) mixed-semantics transformation.

To mitigate this problem, the front-end provides a transformation based on the concept of *mixed-semantics*[2] [63,77]. It relies on the simple observation that any call to a procedure $P$ either fails inside the call and therefore $P$ does not return, or returns successfully from the call. Based on this, any call to $P$ can be instrumented as follows:

- if $P$ may fail, then make a copy of $P$'s body (in main) and jump to the copy.
- if $P$ may succeed, then make the call to $P$ as usual. Since $P$ is known not to fail each assertion in $P$ can be safely replaced with an *assume*.

Upon completion, only the main function has assertions and each procedure is inlined at most once. The explanation for the latter is that a function call is inlined only if it fails and hence, its call stack can be ignored. Mixed-semantics transformation preserves reachability and non-termination properties [63]. Since this transformation is not very common in other verifiers, we illustrate it on an example.

**Example 1 (Mixed-semantics transformation)** *On the left in Figure 2 we show a small program consisting of a main procedure calling two other procedures $p1$ and $p2$ with three assertions $c1$, $c2$, and $c3$. On the right, we show the new program after the mixed-semantics transformation. First, when main calls $p1$ it is transformed into a non-deterministic choice between (a) jumping into the entry block of $p1$ or (b) calling $p1$. The case (a) represents the situation when $p1$ fails and it is done by inlining the body of $p1$ (labeled by $p1_{entry}$) into main and adding a* goto *statement to $p1_{entry}$. The case (b) considers the case when $p1$ succeeds and hence it simply duplicates the function $p1$ but replacing all the assertions with assumptions since no failure is possible. Note that while $p1$ is called twice, it is inlined only once. Furthermore, each inlined function ends up with an "*assume *(false)" indicating that execution dies. Hence, any complete execution of a transformed program corresponds to a bad execution of the original one. Finally, an interesting side-effect of mixed-semantics is that it can provide some context-sensitivity to context-insensitive intra-procedural analyses.*

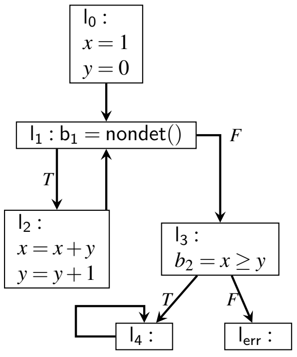
## 3. Verification Conditions

SEAHORN provides out-of-the-box verification semantics with different degrees of abstraction. Furthermore, to accommodate a variety of applications, SEAHORN is designed to be easily extended with a custom semantics as well. In this section, we illustrate the various dimensions of semantic flexibility present in SEAHORN.

[2]The semantics is called *mixed* because it combines small- and big-step operational semantics.

```
int x = 1;
int y = 0;
while (*) {
    x = x + y;
    y = y + 1;
}
assert(x ≥ y);
```

(a)



(b)

$\langle 1\rangle\ \mathsf{p}_0.$

$\langle 2\rangle\ \mathsf{p}_1(x,y) \leftarrow \mathsf{p}_0, x = 1, y = 0.$

$\langle 3\rangle\ \mathsf{p}_2(x,y) \leftarrow \mathsf{p}_1(x,y)\ .$

$\langle 4\rangle\ \mathsf{p}_3(x,y) \leftarrow \mathsf{p}_1(x,y)\ .$

$\langle 5\rangle\ \mathsf{p}_1(x',y') \leftarrow \mathsf{p}_2(x,y), x' = x+y, y' = y+1.$

$\langle 6\rangle\ \mathsf{p}_4 \leftarrow (x \geq y), \mathsf{p}_3(x,y).$

$\langle 7\rangle\ \mathsf{p}_{\mathsf{err}} \leftarrow (x < y), \mathsf{p}_3(x,y).$

$\langle 8\rangle\ \mathsf{p}_4 \leftarrow \mathsf{p}_4.$

(c)

**Figure 3.** (a) Program, (b) Control-Flow Graph, and (c) Verification Conditions.

*Encoding Verification Conditions.* SEAHORN is parametric in the semantics used for VC encoding. It provides two different semantics encodings: (a) a small-step encoding (exemplified in Figure 3) and (b) a large-step encoding. Large-step encoding is similar to the Large Block Encoding (LBE) of [6]. A user can choose the encoding depending on the particular application. In practice, large-step is often more efficient but small-step might be more useful if a fine-grained proof or counterexample is needed. For example, SEAHORN used the large-step encoding in SV-COMP [59].

Regardless of the encoding, SEAHORN uses Constrained Horn Clauses (CHC) to encode the VCs. Given the sets $\mathcal{F}$ of function symbols, $\mathcal{P}$ of predicate symbols, and $\mathcal{V}$ of variables, a *Constrained Horn Clause (CHC)* is a formula in First Order Logic of the following form:

$$\forall \mathcal{V} \cdot (\phi \wedge p_1[X_1] \wedge \cdots \wedge p_k[X_k] \rightarrow h[X]),\ \text{for } k \geq 0$$

where $\phi$ is a constraint over $\mathcal{F}$ and $\mathcal{V}$ with respect to some background theory $T$; $X_i, X \subseteq \mathcal{V}$ are (possibly empty) vectors of variables; $p_i[X_i]$ is an application $p(t_1, \ldots, t_n)$ of an $n$-ary predicate symbol $p \in \mathcal{P}$ for first-order terms $t_i$ constructed from $\mathcal{F}$ and $X_i$; and $h[X]$ is either defined analogously to $p_i$ or is $\mathcal{P}$-free (i.e., no $\mathcal{P}$ symbols occur in $h$). Here, $h$ is called the *head* of the clause and $\phi \wedge p_1[X_1] \wedge \ldots \wedge p_k[X_k]$ is called the *body*. A clause is called a *query* if its head is $\mathcal{P}$-free, and otherwise, it is called a *rule*. A rule with body true is called a *fact*. We say a clause is *linear* if its body contains at most one predicate symbol, otherwise, it is called *non-linear*. In this paper, we follow the Constraint Logic Programming (CLP) convention of representing Horn clauses as $h[X] \leftarrow \phi, p_1[X_1], \ldots, p_k[X_k]$, by omitting explicit universal quantification, replacing implication by an arrow and conjunction by a comma, and writing clauses as rules with the head on the left.

A set of CHCs is satisfiable if there exists a First Order Logic interpretation $\mathcal{I}$ of the predicate symbols $\mathcal{P}$ such that each constraint $\phi$ is true under $\mathcal{I}$. Without loss of generality, deciding whether a program $\mathcal{A}$ satisfies a safety property $\alpha_{safe}$ is reducible to establishing the (un)satifiability of CHCs encoding the VCs of $\mathcal{A}$. We illustrate the process on the following example. Additional examples of the encoding are available in [10].

**Example 2 (Small-step encoding of VCs using CHCs)** *Figure 3(a) shows a program that increments two variables x and y in a non-deterministic loop. After the loop is executed we would like to prove that x cannot be less than y. Ignoring overflow , it is easy to see that the program is safe since x and y are initially non-negative numbers and x is greater than y. Since the loop increases x by a greater amount than y, at its exit x cannot be smaller than y. Figure 3(b) depicts, its corresponding Control Flow Graph (CFG) and Figure 3(c) shows its VCs encoded as a set of CHCs.*

*The set of CHCs in Figure 3(c) essentially represents the* small-step *operational semantics of the CFG. Each basic block is encoded as a CHC. A basic block label $l_i$ in the CFG is translated into a predicate $\mathsf{p_i}(X_1,\ldots,X_n)$ such that $\mathsf{p_i} \in \mathscr{P}$ and $\{X_1,\ldots,X_n\} \subseteq \mathscr{V}$ is the set of live variables at the entry of block $l_i$. A CHC can model both the control flow and data of each block in a succinct way. For instance, the fact $\langle 1 \rangle$ represents that the entry block $l_0$ is reachable. Clause $\langle 2 \rangle$ describes that if $l_0$ is reachable then $l_1$ should be reachable too. Moreover, its body contains the constraints $x = 1 \wedge y = 0$ representing the initial state of the program. Clause $\langle 5 \rangle$ models the loop body by stating that the control flow moves to $l_2$ from $l_1$ after transforming the state of the program variables through the constraints $x' = x + y$ and $y' = y + 1$, where the primed versions represent the values of the variables after the execution of the arithmetic operations. Based on this encoding, the program in Figure 3(a) is safe if and only if the set of recursive clauses in Figure 3(c) augmented with the query $\mathsf{p_{err}}$ is unsatisfiable. Note that since we are only concerned about proving safety (and not termination) any safe final state can be represented by an infinite loop (e.g., clause $\langle 8 \rangle$)).*

SEAHORN middle-end offers a very simple interface for developers to implement an encoding of the verification semantics that fits their needs. At the core of the SEAHORN middle-end lies the concept of a symbolic store. A *symbolic store* simply maps program variables to symbolic values. The other fundamental concept is how different parts of a program are *symbolically executed*. The small-step verification semantics is provided by implementing a symbolic execution interface that symbolically executes LLVM instructions relative to the symbolic store. This interface is automatically lifted to large-step semantics as necessary.

*Modeling statements with different degrees of abstraction.* The SEAHORN middle-end includes verification semantics with different levels of abstraction. Those are, from the coarsest to the finest:

**Registers only:** only models LLVM numeric registers. In this case, the constraints part of CHC is over the theory of Linear Integer Arithmetic (LIA).

**Registers + Pointers (without memory content):** models numeric and pointer registers. This is sufficient to capture pointer arithmetic and determine whether a pointer is `NULL`. Memory addresses are also encoded as integers. Hence, the constraints remain over LIA.

**Registers + Pointers + Memory:** models numeric and pointer registers and the heap. The heap is modeled by a collection of non-overlapping arrays. The constraints are over the combined theories of arrays and LIA.

*Memory encoding.* For concrete semantics, SEAHORN assumes object-based memory model. Each allocation site (heap, stack, and globals) returns a memory object representing a sequence of bytes. The objects are disjoint. Each pointer points to some object and pointer arithmetic is restricted to stay within an object.

For verification condition, an abstract memory model is used that restricts the number of memory objects to be finite. As usual, an abstract memory object represents all concrete objects allocated at a given syntactic allocation site. The memory is further partitioned into regions, where a region is one or more memory objects, such that each memory instruction uses or modifies exactly one memory region. The abstract memory model is context-sensitive – each procedure has its own memory regions.

Memory regions are computed statically using a specialized context-sensitive alias analysis call SEADSA [60]. As the name suggests, SEADSA is a variant of *Data Structure Analysis (DSA)* [79]. DSA itself is an extension of Steensgaard's (a.k.a. unification-based) pointer analysis [97].

In SEADSA, the memory is partitioned into a heap, a stack, and global objects. The analysis builds for each function a *DS graph* where each node represents an abstract memory region. Distinct *nodes* express disjoint sets of memory objects. Edges in the graph represent *points-to* relationships between nodes. Each node is typed and determines the number of fields and outgoing edges in a node. A node can have one outgoing edge per field, but each field can have at most one outgoing edge. This restriction is key to scalability and it is preserved by *unifying* nodes whenever it is violated.

Given a DS graph, each node is mapped to an array in the VC. Then, each memory read (`load`) and write (`store`) in LLVM bitcode is associated with a unique node (i.e., the array). For memory writes, SEAHORN creates a new array variable representing the new state of the array after the write operation.

*Inter-procedural proofs.* For most real programs verifying a function separately from each possible caller (i.e., *context-sensitivity*) is necessary for scalability. The version of SEAHORN for SV-COMP 2015 [59] achieved full context-sensitivity by inlining all program functions. Although inlining is often an effective solution for small and medium-size programs it is well known that suffers from an exponential blow up in the size of the original program. Even more importantly, inlining cannot produce inter-procedural proofs nor counterexamples which are often highly desired.

We tackled this problem in [58], by providing an encoding that allows inter-procedural proofs. We illustrate this procedure via an example in Figure 4. The upper box shows a program with three procedures: *main*, *foo*, and *bar*. The program swaps two numbers $x$ and $y$. The procedure *foo* adds two numbers and *bar* subtracts them. At the exit of *main* we want to prove that the program indeed swaps the two inputs. To show all relevant aspects of the inter-procedural encoding we add a trivial assertion in *bar* that checks that whenever $x$ and $y$ are non-negative the input $x$ is greater or equal than the return value.

The lower box of Figure 4 illustrates the corresponding verification conditions encoded as CHCs. The new encoding follows a small-step style as the intra-procedural encoding shown in Figure 3 but with two major distinctions. First, notice that the CHCs are not linear anymore (e.g., the rule denoted by $\mathsf{m}_{\mathsf{assrt}}$). Each function call has been replaced with a *summary rule* ($\mathsf{f}$ and $\mathsf{b}$) representing the effect of calling to the functions *foo* and *bar*, respectively. The second difference is how assertions are encoded. In the intra-procedural case, a program is unsafe if the query $\mathsf{p}_{\mathsf{err}}$ is satisfiable, where $\mathsf{p}_{\mathsf{err}}$ is

```
$main()$
    $x = \mathsf{nondet}();$
    $y = \mathsf{nondet}();$
    $x_{old} = x;$
    $y_{old} = y;$
    $x = foo(x,y);$
    $y = bar(x,y);$
    $x = bar(x,y);$
    assert $(x = y_{old} \wedge y = x_{old});$
```

```
$foo(x,y)$
    $res = x + y;$
    return $res;$
$bar(x,y)$
    $res = x - y;$
    assert $(\neg\,(x \geq 0 \wedge y \geq 0 \wedge x < res));$
    return $res;$
```

```
$\mathsf{m}_{\mathsf{entry}}.$
$\mathsf{m}_{\mathsf{assrt}}(x_{old}, y_{old}, x, y, e_{out}) \leftarrow$
    $\mathsf{m}_{\mathsf{entry}},$
    $x_{old} = x, y_{old} = y,$
    $\mathsf{f}(x, y, x_1),$
    $\mathsf{b}(x_1, y, y_1, \mathsf{false}, e),$
    $\mathsf{b}(x_1, y_1, x_2, e, e_{out}).$
$\mathsf{m}_{\mathsf{err}}(e_{out}) \leftarrow$
    $\mathsf{m}_{\mathsf{assrt}}(x_{old}, y_{old}, x, y, e), \neg\, e,$
    $e_{out} = \neg\,(x = y_{old}, y = x_{old}).$
$\mathsf{m}_{\mathsf{err}}(e_{out}) \leftarrow$
    $\mathsf{m}_{\mathsf{assrt}}(x_{old}, y_{old}, x, y, e_{out}), e_{out}.$
```

```
$\mathsf{f}_{\mathsf{entry}}(x,y).$
$\mathsf{f}_{\mathsf{exit}}(x, y, res) \leftarrow$
    $\mathsf{f}_{\mathsf{entry}}(x,y),$
    $res = x + y.$
$\mathsf{f}(x, y, res) \leftarrow \mathsf{f}_{\mathsf{exit}}(x, y, res).$
$\mathsf{b}_{\mathsf{entry}}(x,y).$
$\mathsf{b}_{\mathsf{exit}}(x, y, res, e_{out}) \leftarrow$
    $\mathsf{b}_{\mathsf{entry}}(x,y),$
    $res = x - y,$
    $e_{out} = (x \geq 0 \wedge y \geq 0 \wedge x < res).$
$\mathsf{b}(x, y, z, \mathsf{true}, \mathsf{true}).$
$\mathsf{b}(x, y, z, \mathsf{false}, e_{out}) \leftarrow \mathsf{b}_{\mathsf{exit}}(x, y, z, e_{out})$
```

**Figure 4.** A program with procedures (upper) and its verification condition (lower).

the head of a CHC associated with a special basic block to which all can-fail blocks are redirected. However, with the presence of procedures assertions can be located deeply in the call graph of the program, and therefore, we need to modify the CHCs to propagate error to the main procedure.

In our example, since a call to *bar* can fail we add two arguments $e_{in}$ and $e_{out}$ to the predicate b where $e_{in}$ indicates if there is an error before the function is called and $e_{out}$ indicates whether the execution of *bar* produces an error. By doing this, we are able to propagate the error in clause $\mathsf{m}_{\mathsf{assrt}}$ across the two calls to *bar*. We indicate that no error is possible at *main* before any function is called by unifying false with $e_{in}$ in the first occurrence of b. Within a can-fail procedure we skip the body and set $e_{out}$ to true as soon as an assertion can be violated. Furthermore, if a function is called and $e_{in}$ is already true we can skip its body (e.g., first clause of b). Functions that cannot fail (e.g., *foo*) are unchanged. The above program is safe if and only if the query $\mathsf{m}_{\mathsf{err}}(\mathsf{true})$ is unsatisfiable.

Finally, it is worth mentioning that this propagation of error is not required if the mixed-semantics transformation described in Section 2 is applied.

## 4. Verification Engines

In principle, SEAHORN can be used with any Horn clause or LLVM-based verification tool. In the following, we describe two such tools developed by ourselves. Notably, the tools discussed below are based on the contrasting techniques of SMT-based model checking and Abstract Interpretation, showcasing the wide applicability of SEAHORN.

### 4.1. SMT-Based Model Checking with SPACER

SPACER is an efficient SMT-based Model Checker for deciding satisfiability of Constrained Horn Clauses (CHC) [73–75]. Of course, since CHC satisfiability is undecidable, we use the term *decision procedure* informally. SPACER is a sound procedure, but it is not complete (i.e., not formally a decision procedure, and does not terminate on all inputs). In contrast to other SMT-based Model Checking algorithms (for example, those based on based on interpolation [2,53,66,80]), the reasoning in SPACER is compositional (or modular). That is, the *transition relation* is not unrolled. SPACER reasons about a body of individual procedure (or predicate) at a time, and communicates information between procedures (or predicates) using summaries. This is crucial for scaling SMT-based Model Checking to programs. Unlike hardware circuits, an unrolling of a program (i.e., unrolling loops and inlining procedures) increases the size of an SMT formula representing a verification condition (VC) exponentially. The approach taken by SPACER avoids the exponential explosion by limiting the information that can be exchanged between procedures to well-defined summaries. The summaries also provide a form of caching to prevent exploring the same procedure in the same calling context multiple times.

SPACER is integrated into SMT-solver Z3 [34] and is currently the default CHC engine in Z3. It supports CHC with constraints in the (combined) theories of Linear Real Arithmetic [9], Linear Integer Arithmetic [75], Arrays [73], with basic support for theories of Bit Vectors and Abstract Data Types. Both quantifier free and universally quantified solutions for the theory of arrays are supported [61].

In this section, we give a high-level overview of SPACER algorithm. Many implementation details and optimizations are omitted since they often change between different versions of the implementation. SPACER builds on three main concepts: Craig Interpolation, Model Based Projection, and an IC3/PDR-style Model Checking algorithm. In the rest of this section, we describe each component in turn, starting with interpolation.

#### 4.1.1. Craig Interpolation

Let $A$ and $B$ be two formulas in First Order Logic such that $A \wedge B$ is unsatisfiable. A *Craig interpolant* (or simply an interpolant) is a formula $I$ such that

$$A \implies I \qquad\qquad I \implies \neg B$$

and the only uninterpreted constants and functions in $I$ are those that are shared between $A$ and $B$. For example, consider the following two formulas $A$ and $B$ in Linear Integer Arithmetic:

$$A = (a < x \wedge x < b) \qquad\qquad B = (a = 1 \wedge b = 1)$$

The conjunction $A \wedge B$ is unsatisfiable: if $a = b = 1$ then there cannot be an integer $x$ strictly between $a$ and $b$. The shared uninterpreted constants are $a$ and $b$. There is an interpolant, but it is not unique. Several possible interpolants are:

$$I_1 = (a < b) \qquad I_2 = \neg(a = 1 \wedge b = 1) \qquad I_3 = (a \neq b)$$

All of the above formulas $I_i$ (for $1 \leq i \leq 3$) is an interpolant. We write $\textsc{Itp}(A, B)$ for some interpolant between $A$ and $B$ if it exists.

It is well known that interpolants can be computed directly from a resolution refutation of satisfiability of $A$ and $B$. In case of SMT, interpolation is a combination of interpolation over propositional resolution [62] and special procedures for theory lemmas and their derivation [19,54]. We refer the reader to the references above for more details.

In the case of SPACER, the interpolation problem is more restricted and simplified. We are only interested in computing an interpolant $\textsc{Itp}(A,B)$ in the case where $B$ is a conjunction of literals and every uninterpreted symbol of $B$ is shared with $A$. In this case, the simplest choice for an $\textsc{Itp}(A,B)$ is $\neg B$. Since $A \wedge B$ is unsatisfiable, it follows that $A \implies \neg B$, and obviously $\neg B$ implies itself. Note that in the example above, $I_2$ is simply $\neg B$. On one hand, using $\neg B$ as an interpolant defeats the purpose of interpolation. On the other, it provides a default case, that is often avoided, but is possible when a different interpolant is hard to compute. This is especially convenient for a system like Z3 that does not consistently produce easy-to-interpolate proofs.

Under the restrictions above, another alternative for an interpolant a negation of a Minimal Unsatisfiable Subset (MUS) of $B$. The reasoning is the same as for using $\neg B$. Such an interpolant does not do much generalization, but might filter irrelevant facts.

In practice, interpolant computation used by SPACER is a mix of proof-based and MUS-based procedure. A proof, and, in particular, theory lemmas of the proof, are examined to extract the interaction of $B$ literals with the refutation. If the interaction is fairly clear, an *interpolating unsat core* is extracted by using interpolation-style reasoning. If the interaction is not clear, an MUS for $B$ is computed. This style of reasoning enables SPACER to construct an interpolants such as $I_1$, $I_2$, and $I_3$ in the example above. However, the exact interpolant constructed depends on the proof produced by Z3.

### *4.1.2. Model-Based Projection*

Let $\varphi$ be a satisfiable formula with uninterpreted constants (or variables) $\mathit{Vars}(\varphi)$. Let $U$ be a subset of variables in $\mathit{Vars}(\varphi)$, and $M \models \varphi$ be a model of $\varphi$. A formula $\psi$ is a *Model Based Projection* (MBP) of $U$ relative to $M$ iff (a) $M \models \psi$, (b) $\psi \implies \exists U \cdot \varphi$, (c) $\mathit{Vars}(\psi) \subseteq \mathit{Vars}(\varphi) \setminus U$, and (d) $\psi$ is a monomial (i.e., a conjunction of literals). Intuitively, MBP under-approximates projection (or quantifier elimination). Alternatively, MBP $\psi$ can be seen as a generalization of the model $M$: $\psi$ *contains* the model, yet, it is a formula (i.e., has a finite representation) and is contained in the projection $\exists U \cdot \varphi$.

We write $\textsc{Mbp}(\exists U \cdot \varphi, M)$ for an MBP procedure that given an existentially quantified formula and a model, returns a corresponding model-based projection. An MBP procedure is *finite* if it is finite in the model argument. That is, the function $\lambda x \cdot \textsc{Mbp}(\exists U \cdot \varphi, x)$ has a finite range. It is not difficult to see that a theory that admits quantifier elimination has a finite MBP. Consider a formula $\exists U \cdot \varphi$. Assume that there is an equivalent quantifier free formula $\psi$:

$$\psi \iff \exists U \cdot \varphi$$

Let $\psi_1 \vee \cdots \vee \psi_n$ be a DNF decomposition of $\psi$. Then, define $\textsc{Mbp}(\exists U \cdot \varphi, M) = \psi_i$ such that $i = \min\{1 \leq j \leq n \mid M \models \psi_j\}$.

Conversely, a finite MBP provides a procedure for quantifier elimination. Let $M_1$ be a model for $\varphi$, and $\psi_1 = \textsc{Mbp}(\exists U \cdot \varphi, M_1)$. Let $M_2$ be a model for $\varphi \wedge \neg \psi_1$, and $\psi_2 = \textsc{Mbp}(\exists U \cdot \varphi, M_2)$, etc. Since by assumption MBP is finite, the number of such $\psi_i$

is finite as well. Hence the formula $\psi_1 \vee \cdots \vee \psi_n$ is well defined, quantifier free, and is equivalent to $\exists U \cdot \varphi$.

A finite MBP for Linear Real Arithmetic has been introduced in [74]. Unlike quantifier elimination for LRA, it can be computed in linear time (assuming the model is given and evaluating literals in the model is constant time). A finite MBP for Linear Integer Arithmetic and Abstract Data Types has been presented in [11]. MBP for the theory of Arrays has been developed in [73]. Obviously, MBP for arrays is not finite.

We illustrate an MBP procedure for the combined theories of arrays and arithmetic using an example below. Let $\varphi$ denote the formula

$$(b = a[i_1 \leftarrow v_1]) \vee (a[i_2 \leftarrow v_2][i_3] > 5 \wedge a[i_4] > 0)$$

where $a$ and $b$ are array variables whose index and value sorts are both Int, the sort of integers, and all other variables have sort Int. Here, for an array $a$, we use $a[i \leftarrow v]$ to denote a *store* of $v$ into $a$ at index $i$ and use $a[i]$ to denote the value of $a$ at index $i$. Suppose that we want to existentially quantify the array variable $a$. Let $M \models \varphi$. We will consider two possibilities for $M$:

1. Let $M \models b = a[i_1 \leftarrow v_1]$, i.e., $M$ satisfies the array equality containing $a$. In this case, our MBP procedure substitutes the term $b[i_1 \leftarrow x]$ for $a$ in $\varphi$, where $x$ is a fresh variable of sort Int. That is, the result of MBP is $\exists x \cdot \varphi[b[i_1 \leftarrow x]/a]$.
2. Let $M \models b \neq a[i_1 \leftarrow v_1]$. We use the second disjunct of $\varphi$ for MBP. Furthermore, let $M \models i_2 \neq i_3$. We then reduce the term $a[i_2 \leftarrow v_2][i_3]$ to $a[i_3]$ to obtain $a[i_3] > 5 \wedge a[i_4] > 0$, using the relevant disjunct of the select-after-store axiom of ARR. We then introduce fresh variables $x_3$ and $x_4$ to denote the two select terms on $a$, obtaining $x_3 > 5 \wedge x_4 > 0$. Finally, we add $i_3 = i_4 \wedge x_3 = x_4$ if $M \models i_3 = i_4$ and add $i_3 \neq i_4$ otherwise, choosing the relevant case of Ackermann reduction, and existentially quantify $x_3$ and $x_4$.

Model-Based Projection is crucial for SPACER. It is used both in computing predecessors and summaries. However, since its inception, it has found many other applications as well. For example, in [11] it is used in a procedure for deciding satisfiability of quantified formulas. In [40] to discover a simulation relation between different version of a program. In [72] it is extended with Skolemization and is used to synthesize implementation from assume-guarantee contracts. The Skolemization procedure is further improved in [39].

### *4.1.3.* SPACER *Algorithm*

Without loss of generality, we assume that set of CHCs encoding safety of procedural programs is transformed into an equisatisfiable set of just *three* clauses with a *single* predicate symbol of the following form:

$$\begin{gathered} Inv(\bar{x}) \leftarrow Init(\bar{x}) \qquad \neg Bad(\bar{x}) \leftarrow Inv(\bar{x}) \\ Inv(\bar{x}') \leftarrow Inv(\bar{x}), Inv(\bar{x}^o), Tr(\bar{x}, \bar{x}^o, \bar{x}') \end{gathered} \tag{1}$$

The notation $\bar{x}^\dagger$ stands for a vector of variables obtained from $\bar{x}$ by adding $^\dagger$ to every variable, where $\dagger \in \{', ^o\}$. For example, $(x_1, x_2, x_3)'$ is $(x_1', x_2', x_3')$. Intuitively, *Inv* is the program invariant, $\bar{x}$ denotes the pre-state of a program transition, $\bar{x}'$ denotes the post-state, and $\bar{x}^o$ denotes the summary of a procedure call (if one is made). Any verification condition for sequential programs can be transformed into the form of (1) by adding extra state variables that denote active program location and active procedure being executed.

In the special case of verification conditions of procedure-free sequential programs, $\bar{x}^o$ variables do not appear in *Tr* and the conjunct $Inv(\bar{x}^o)$ can be dropped. The resulting three clauses simplify to the following:

$$\begin{gathered} Inv(\bar{x}) \leftarrow Init(\bar{x}) \qquad \neg Bad(\bar{x}) \leftarrow Inv(\bar{x}) \\ Inv(\bar{x}') \leftarrow Inv(\bar{x}), Tr(\bar{x},\bar{x}') \end{gathered} \tag{2}$$

In the case of (2), *Inv* denotes a regular inductive invariant of a transition system.

To simplify the notation, we introduce a special function $\mathscr{F}_{Tr}$, called a *forward transformer*, that replaces *Inv* in the rule by a pair of formulas $\phi_A(\bar{x})$ and $\phi_B(\bar{x})$. Formally, it is defined as follows:

$$\mathscr{F}_{Tr}(\varphi_A, \varphi_B) \quad \equiv \quad Init(\bar{x}') \vee \left(\varphi_A(\bar{x}) \wedge \varphi_B(\bar{x}^o) \wedge Tr(\bar{x},\bar{x}^o,\bar{x}')\right)$$

Abusing notation, we write $\mathscr{F}_{Tr}(\varphi_A)$ for $\mathscr{F}_{Tr}(\varphi_A, \varphi_A)$, and $\mathscr{F}(\varphi_A, \varphi_B)$ when *Tr* is clear from the context or is irrelevant. Using function $\mathscr{F}$, the CHC in (1) are equivalently expressed as two First Order Logic formulas:

$$\forall \bar{x},\bar{x}',\bar{x}^o \cdot \mathscr{F}(Inv,Inv) \implies Inv(\bar{x}') \qquad\qquad \forall \bar{x} \cdot Inv(\bar{x}) \implies \neg Bad(\bar{x})$$

SPACER is a parameterized algorithm (or a set of rules) that is instantiated for a given logical theory $T$ given three ingredients: (a) a model-producing satisfiability solver for $T$ (i.e., an SMT solver that supports theory $T$), (b) an MBP procedure MBP for $T$, and (c) an interpolation procedure ITP for $T$. Here, we present a version that is limited to quantifier free solutions. Extension of SPACER for quantified solutions is described in [61].

The main data-structures operated by SPACER are a sequence of *may* summaries $[F_0, F_1, \ldots]$ called a *trace*, a *must* summary called $R$, and a queue of proof obligations $Q$. Each element $F_i$ of a trace is called a frame, and each element $\ell \in F_i$ is called a lemma (or a may summary). Intuitively, $F_i$ over-approximates all the states reachable by *Tr* in up to $i$ steps (derivations). The set $R$, also called the *reachable states*, under-approximates all the reachable states. Finally, elements of $Q$, called proof-obligations, represent states the algorithm is trying to proof reachable or unreachable.

The rules defining SPACER are shown in Alg. 1. The rules are applied non-deterministically although, only some order of application guarantees progress. Each rule is presented as a guarded command "[ *grd* ] *cmd*", where *cmd* can be executed only if *grd* holds. If multiple guards are true, any one of the corresponding commands can be executed.

As described above, SPACER maintains a set of reachability queries $Q$, a sequence of may summaries $\{F_i\}_{i \in \mathbb{N}}$, and a must summary $R$. Intuitively, a query $\langle \varphi, i \rangle$ corresponds to checking if $\varphi$ is reachable for recursion depth $i$, $F_i$ over-approximates the reachable states for recursion depth $i$, and $R$ under-approximates the reachable states. $N$ denotes the current bound on recursion depth. The sequence of may summaries and $N$ correspond

```
Input: Formulas $Init(\bar{x}), Tr(\bar{x}, \bar{x}^o, \bar{x}'), Bad(\bar{x})$
Output: An inductive invariant or UNSAFE

if $(Init \wedge Bad)$ satisfiable then return UNSAFE
// initialize data structures
$Q := \emptyset$                                        // set of pairs $\langle \varphi, i \rangle, i \in \mathbb{N}$
$N := 0$                                                // max level, or recursion depth
$F_0 = Init, F_i = \top, \forall i > 0$                 // may summary sequence
$R = Init$                                              // must summary
forever non-deterministically do
    (Candidate) [ $(F_N \wedge Bad)$ satisfiable ]
        $Q := Q \cup \langle \varphi, N \rangle$, for some $\varphi \implies F_N \wedge Bad$
    (MustPredecessor) [ $\langle \varphi, i+1 \rangle \in Q$, $M \models \mathscr{F}(F_i, R) \wedge \varphi'$ ]
        $Q := Q \cup \langle \textsc{Mbp}(\exists \bar{x}^o, \bar{x}' \cdot \mathscr{F}(F_i, R) \wedge \varphi', M), i \rangle$
    (MayPredecessor) [ $(\varphi, i+1) \in Q$, $M \models \mathscr{F}(F_i) \wedge \varphi'$ ]
        $Q := Q \cup \langle \textsc{Mbp}(\exists \bar{x}, \bar{x}' \cdot \mathscr{F}(F_i) \wedge \varphi', M)[\bar{x}/\bar{x}^o], i \rangle$
    (Leaf) [ $(\varphi, i) \in Q$, $\mathscr{F}(F_{i-1}) \implies \neg\varphi'$, $i < N$ ]
        $Q := Q \cup \langle \varphi, i+1 \rangle$
    (Successor) [ $\langle \varphi, i+1 \rangle \in Q$, $M \models \mathscr{F}(R) \wedge \varphi'$ ]
        $R := R \vee \textsc{Mbp}(\exists \bar{x}, \bar{x}^o \cdot \mathscr{F}(R) \wedge \varphi', M)[\bar{x}/\bar{x}']$
    (NewLemma) [ $\langle \varphi, i+1 \rangle \in Q$, $\mathscr{F}(F_i) \implies \neg\varphi'$ ]
        $F_j := F_j \wedge \textsc{Itp}(\mathscr{F}(F_i), \varphi')[\bar{x}/\bar{x}'], \forall j \leq i+1$
    (Induction) [ $(\varphi \vee \psi) \in F_i$, $\mathscr{F}(\varphi \wedge F_i) \implies \varphi'$ ]
        $F_j := F_j \wedge \varphi, \forall j \leq i+1$
    (Unfold) [ $F_N \implies \neg Bad$ ] $N := N+1$
    (Safe) [ $F_{i+1} \implies F_i$ ] return $F_i$
    (Unsafe) [ $(R \wedge Bad)$ satisfiable ] return UNSAFE
```

**Algorithm 1:** Rule-based description of SPACER.

to the *trace of approximations* and the maximum *level* in IC3/PDR, respectively. For convenience, let $F_{-1}$ be $\perp$. $\textsc{Mbp}(\varphi, M)$, for a formula $\varphi = \exists \bar{v} \cdot \varphi_{qf}$ and model $M \models \varphi_{qf}$, denotes the result of some MBP function associated with $\varphi$ for the model $M$.

Alg. 1 initializes $N$ to 0 and, $F_0$ and $R$ to *Init*. **Candidate** initiates a backward search for a counterexample beginning with a set of states in *Bad*. The potential counterexample is expanded using either **MustPredecessor** or **MayPredecessor**. **MustPredecessor** *jumps over* the call $Inv(\bar{x}^o)$, in the last CHC of (1), utilizing the must summary $R$. **MayPredecessor**, on the other hand, creates a query for the call using the may summary of its calling context. **Leaf** moves an unreachable query to a higher recursion depth. **Successor** updates $R$ when a query is known to be reachable. **NewLemma** updates may summaries when a query is known to be unreachable. **Induction** strengthens may summaries using induction relative to $F_i$. **Unfold** increments the bound on the recursion depth. **Safe** returns $F_i$ as invariant when the sequence of may summaries converges. **Unsafe** applies when the must summary intersects with *Bad*.

SPACER is sound and if MBP utilizes finite MBP functions, SPACER also terminates for a fixed $N$. Soundness follows from the fact that the following invariants are maintained by the main loop:

$$Init \implies F_0 \qquad \forall 0 < i \leq N \cdot \mathscr{F}(F_{i-1}) \implies F_i$$

$$R \implies \mathscr{F}^N(Init) \qquad \forall 0 < i \leq N \cdot F_{i-1} \implies F_i$$

$$\forall 0 < i \leq N \cdot \mathscr{F}^i(Init) \implies F_i$$

Thus, $\{F_i\}_{i \in \mathbb{N}}$ and $R$, respectively, over- and under-approximate reachable states.

The rules in Alg. 1 leave out many important implementation details. For efficiency, queries are restricted to cubes (i.e., conjunction of literals). For Linear Arithmetic, the implementation relies on the fact that MBP is linear in time and space. $Q$ is maintained as a priority queue, processing queries of smaller recursion depths first. Additional constraints are imposed on the rules and their ordering to ensure termination for a fixed $N$. For the rule **Unsafe**, the implementation also produces a counterexample in addition to returning UNSAFE.

## 4.2. *Abstract Interpretation with* CRAB

In this section, we first introduce CRAB [31] (CoRnucopia of ABstractions), a language-agnostic static analyzer based on the theory of Abstract Interpretation [25]. CRAB does not analyze directly LLVM bitcode but instead it analyzes a goto-based Control-Flow Graph language. This allows decoupling the analyzer from the input language so that it can be reused for analyzing other languages beyond LLVM bitcode (e.g., in [48]). Then, we describe CLAM (CRAB for Llvm Abstraction Manager), the static analyzer of LLVM bitcode based on CRAB, which is integrated in SEAHORN as one of its back-end solvers.

Note that we have decided to implement CRAB on top of a imperative language and not directly on Constrained Horn Clauses. This is motivated by the necessity to orient the analysis in Abstract Interpretation. That is, the interpreter needs to know the order in which to execute the instructions. While it is possible to map logical definitions into instructions, in practice, we chose to avoid this complication by going directly from LLVM to the intermediate representation used by CRAB.

### 4.2.1. CRAB *Target Language*

CRAB programs are written in the goto-based language described in Figure 5. A program $P$ consists of a non-empty sequence of basic blocks, each one denoted by a unique identifier *bb*, containing zero or more instructions $I$ in three-address form. Operands can only be one of these three basic types: integers, booleans, and pointers, or arrays of a basic type. All instructions are strongly typed. The language does not support floating point operations.

Integer, boolean, pointer, and array variables are denoted with symbols $v_i$, $v_b$, $v_p$, $v_A$, respectively. Variables of any type are denoted by $v$. Scalar (non-array) variables are denoted by $v_s$. Integer variables are sized (i.e., of different bit-width). The set of integer, boolean, pointer, and array variables are disjoint.

$$
\begin{array}{lcl}
P & ::= & B^{+} \\
B & ::= & bb : I^{*}\ \textbf{goto}\ bb_1, \ldots bb_n \ \mid\ bb : I^{*}\ [\ \textbf{return}\ v_1, \ldots, v_n\ ] \\
I & ::= & I_a \mid I_b \mid I_p \mid I_A \mid v'_1, \ldots, v'_m := fun(v_1, \ldots, v_n) \\
 & & v := \textbf{havoc}\ () \mid \textbf{assume}\ (v_b) \mid \textbf{assume}\ (b) \mid \textbf{assert}\ (v_b) \mid \textbf{assert}\ (b) \\
I_a & ::= & v_i := a \\
 & & v_i := \textbf{sign\_extension}\ (v_i) \mid v_i := \textbf{zero\_extension}\ (v_i) \mid v_i := \textbf{truncate}\ (v_i) \\
 & & v_i := \textbf{booltoint}\ (v_b) \\
I_b & ::= & v_b := b \mid v_b := \textbf{inttobool}\ (v_i) \\
I_p & ::= & v_p := p \mid v_p := \textbf{alloc}\ (sz) \mid v_s := \textbf{load}\ (v_p) \mid \textbf{store}\ (v_s, v_p) \mid v_p := \&\text{fun} \\
I_A & ::= & v_A := \textbf{array\_init}\ (v_i, v'_i, v_s, sz, endian) \mid v_s := \textbf{array\_select}\ (v_A, v_i, sz, endian) \\
 & & \textbf{array\_write}\ (v_A, v_i, v_s, sz, endian) \mid v'_A := v_A \\
a & ::= & n \mid v_i \mid a_1\ op_a\ a_n \\
b & ::= & \mathsf{true} \mid \mathsf{false} \mid \neg\, b \mid b_1\ op_b\ b_2 \mid a_1\ op_r\ a_2 \mid p_1\ op_p\ p_2 \\
p & ::= & \mathsf{null} \mid v_p + a
\end{array}
$$

**Figure 5.** CRAB goto-based language to represent Control Flow Graphs.

*Arithmetic and boolean instructions.* Arithmetic and boolean expressions are described by $a$ and $b$. CRAB supports standard operations $op_a$ and $op_b$ for these expressions. For arithmetic expressions, CRAB supports addition, subtraction, multiplication, signed/unsigned division, signed/unsigned remainder, and standard bitwise operations: `and`, `or`, `xor`, left shift, logical and arithmetic right shift. For boolean expressions, CRAB supports the operations `and`, `or`, and `xor`.

*Control flow and assertions.* Control flow is modeled by **goto** and **assume** instructions. The instruction $v := \textbf{havoc}()$ assigns non-deterministically any value allowed by v's type to $v$. Properties can only be defined by adding **assert** instructions.

*Pointer instructions.* The instruction $v_p := \textbf{alloc}(sz)$ allocates a fresh memory object of size $sz$ and returns a pointer to it. The instructions $v_s := \textbf{load}(v_p)$ and $\textbf{store}(v_s, v_p)$ read and write memory. Pointer arithmetic can be expressed by $v_p := v'_p + a$. CRAB also supports function pointers $v_p := \&\text{fun}$. For pointer comparisons $op_p$, CRAB supports pointer equality and disequality.

*Array instructions.* CRAB language supports unidimensional arrays. The importance of arrays is inherited from the importance of arrays in imperative languages and even more important, because the program memory can be modeled as an array. Arrays are interpreted as sequences of consecutive bytes which are disjoint from each other. We describe informally the semantics of the array operations. We define first $\mathsf{BS}(v_A, v_i, sz, endian)$ as the byte sequence:

$$
\begin{cases}
v_A[v_i] \cdot v_A[v_i + 1] \cdots v_A[v_i + sz - 1] & \text{if } endian = \mathsf{big} \\
v_A[v_i + sz - 1] \cdot v_A[v_i + sz - 2] \cdots v_A[v_i] & \text{if } endian = \mathsf{little}
\end{cases}
$$

Similarly, we define $\mathsf{BS}(v_s, endian)$ as the byte sequence:

$$
\begin{cases}
v_s(0) \cdots v_s(n-1) & \text{if } endian = \mathsf{big} \\
v_s(n-1) \cdots v_s(0) & \text{if } endian = \mathsf{little}
\end{cases}
$$

The instruction $v_A$ := **array_init** $(v_i, v_i', v_s, sz, endian)$ creates a fresh array $v_A$ such that for all $v_i \leq j < v_i' \wedge (j \bmod sz = 0)$, each byte sequence $\mathsf{BS}(v_A, j, sz, endian)$ is equal to $\mathsf{BS}(v_s, endian)$. Array reads $v_s$ := **array_select** $(v_A, v_i, sz, endian)$ assigns the byte sequence $\mathsf{BS}(v_A, v_i, sz, endian)$ to $\mathsf{BS}(v_s, endian)$. **array_write** $(v_A, v_i, v_s, sz, endian)$ writes the byte sequence $\mathsf{BS}(v_s, endian)$ into $\mathsf{BS}(v_A, v_i, sz, endian)$. Finally, $v_A' := v_A$ assigns all contents of $v_A$ to $v_A'$.

Endianess can be optionally provided. However, if it is not available then array abstract domains are limited when reasoning about byte aliasing because they cannot make any assumption about endianess.

*Function calls.* CRAB assumes call-by-value parameter passing. Functions can return multiple values. This is specially useful for purifying functions. *Function purification* converts functions into new equivalent functions that have no side effects.

*Conversion between types.* Conversion between operand types is allowed but it must be done through explicit casts. CRAB supports sign and zero extension, truncation, and conversions between boolean and integers. Conversion between integers and pointers is not currently supported.

CRAB *language design choices.* The design of the CRAB language has been carefully chosen based on our experience in building abstract interpreters and front-ends. For instance, the language distinguishes between boolean and integer variables although boolean can be also modeled as integers if desired. The distinction between boolean and integers can make easier the translation to the CRAB language if the front-end already makes that separation. Moreover, it can simplify the code of an abstract interpreter because boolean and arithmetic instructions can be analyzed by different abstract domains: boolean instructions with a finite domain and numerical instructions with a numerical abstract domain. The distinction between pointer and arrays instructions is another good example. Typically, abstract domains reasoning about pointers (e.g., [85, 100]) are very different from domains reasoning about arrays (e.g., [29, 49, 65]). The former focuses on aliasing while the latter focuses more on the problem of weak versus strong updates. Again, having specialized instructions for pointers and arrays can simplify the code of the abstract interpreter. Moreover, there are situations where either the input language is simple enough that aliasing is not an issue (e.g., [48]) or the front-end can resolve aliasing at translation time (see Section 4.2.7). For those cases, array domains are sufficient to reason precisely about memory.

Compared to other intermediate representations such as LLVM IR, control flow is expressed in a more declarative way by having **goto** and **assume** instead of conditional branches. Unlike LLVM IR, CRAB language is not intended to be executed, and thus, it allows expressing non-determinism through **havoc** instructions which is very useful for program abstractions (e.g., model a cast from an integer to a pointer). Another key difference with LLVM IR is that the CRAB language does not require the input program to be written in Static Single Assignment (SSA), and, therefore, it does not have $\phi$-nodes. This special instruction is used to represent all the possible values of a variable can take at a merge point in the Control Flow Graph (CFG). The analysis of $\phi$-nodes using abstract interpretation is specially challenging with relational numerical domains [43]. For that reason, Crab language does not allow $\phi$-nodes.

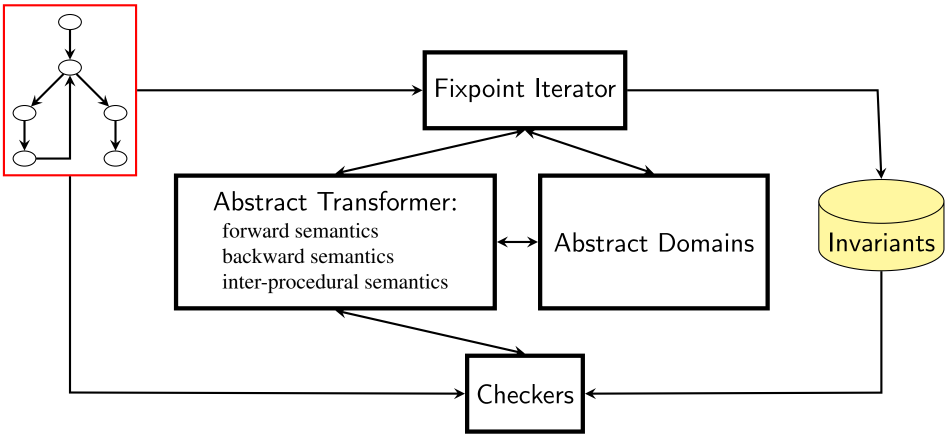


**Figure 6.** CRAB architecture.

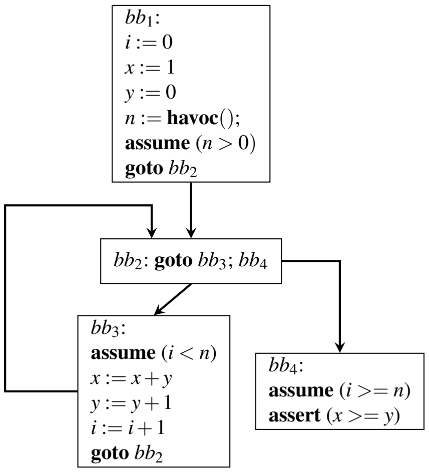
### *4.2.2. Tool Architecture*

The design of Crab is very similar to other abstract interpreters such as ASTRÉE [14], CLOUSOT [38] and IKOS [16]. Crab is parametric both in the fixpoint iterators and abstract domains. The main architecture of the tool is depicted in Figure 6. We omit for now the details about how to adapt this architecture to inter-procedural analysis. This is described in in Section 4.2.6. Crab has two operation modes: the *inference mode* and the *checking mode*.

*Inference mode.* CRAB takes as input the CFG as described in previous section. Then, it solves iteratively the semantic equations extracted directly from the CFG. Solving these equations is performed by the Fixpoint Iterator. The fixpoint iterator is in charge of finding good iteration strategies and in charge of applying widening and narrowing in effective ways, while optimizing time and memory consumption. Solving semantic equations requires to both interact with Abstract Domains (e.g., join, meet, widening, and narrowing) and with Abstract Transformer to apply the corresponding semantics to each CFG instruction (e.g., forward semantics, backward semantics, or inter-procedural semantics). Figure 7 shows an example of semantic equations extracted from a sample CFG.

Finally, after Fixpoint Iterator has found a stable solution (a.k.a. invariants), these are stored in an *invariant database*. When inference mode is enabled, no warning is displayed when some possible error is detected. This is because either the fixpoint has not been reached yet and it might be unsound to report the program is safe, or a warning can be ruled out after refinement using narrowing or dual narrowing operators [23, 28].

*Checking mode.* Upon completion of the inference phase, CRAB uses the invariants stored in the database to check if certain properties can be violated, issuing warning messages. This is done by Checkers. Currently, CRAB can perform some built-in checks (division by zero, null-dereference, etc) and user-defined assertions (**assert** instructions). For efficiency, invariants are only stored per CFG basic block. Thus, instructions might be re-analyzed but only up to the end of each basic block. This saves us from storing invariants per instruction at a very small cost.



$$\begin{aligned}
X_0 &= \top \\
X_1 &= \mathsf{forget}(n, \mathsf{post}(\{i = 0; x := 1; y := 0\}, \\
&\qquad\qquad X_0)) \sqcap (n > 0) \\
X_2 &= X_2 \nabla (X_1 \sqcup X_3) \\
X_3 &= \mathsf{post}(\{x := x + y; y := y + 1; i := i + 1\}, \\
&\qquad (X_2 \sqcap (i < n))) \\
X_4 &= X_2 \sqcap (i \geq n) \sqcap (x < y)
\end{aligned}$$

**Figure 7.** Crab CFG and its forward semantic equations. Each $X_i$ $(1 \leq i \leq 4)$ represents the invariants that hold at the exit of block $bb_i$. The initial abstract state is $X_0 = \top$ (top). The assertion holds if $X_4 = \bot$ (bottom).

### *4.2.3. Fixpoint Iterator*

CRAB computes first a weak topological ordering (WTO) of the CFG following Bourdoncle's algorithm [15]. The WTO produces a good order in which basic blocks should be analyzed, and the set of basic blocks in the CFG where the fixpoint algorithm needs to apply widening to ensure termination.

The current fixpoint iterator used in Crab is based on [4], and it interleaves widening and narrowing operations for each inner loop until reaching convergence before analyzing outer loops. Thresholds are used to improve the precision of the widening (as in [14, 38]). The thresholds are collected statically from the constants appearing in **assume** and **assert** instructions.

The Fixpoint iterator is very generic since it focuses only on solving semantic equations. The particular semantics is given by the Abstract Transformer. The advantage is that it can be replaced by any method as long as one focuses on iterative solving techniques [50, 51, 64, 76, 83, 94].

### *4.2.4. Abstract Domains*

Abstract domains are in charge of interpreting in the abstract the operators $\sqcup, \nabla, \ldots$, and transfer functions appearing during the solving of semantic equations. A very simplified view of the abstract domain interface in CRAB is shown in Figure 8.

The forward (post) and backward (pre) transfer functions are used by Abstract Transformer, and the Fixpoint iterator calls the other operations while computing the fixpoint of the semantic functions. The inter-procedural semantics is implemented by the Abstract Transformer calling directly abstract domain operations $(\mathsf{project}, \sqcap, \mathsf{forget}, \ldots)$.

*Intervals and Congruences.* *Intervals* [24] expresses constraints of the form $x = [lb, ub]$ meaning that $lb \leq x$ and $x \leq ub$ where $x$ is an integer variable and $lb, ub \in \mathbb{Z} \cup \{-\infty, +\infty\}$.

| |
|---|
| *Constructors* |
| makeTop : $D$<br>makeBottom : $D$ |
| *Forward and backward transfer functions* |
| post : $Instr^+ \times D \mapsto D$ |
| pre : $Instr^+ \times D \mapsto D$ |
| forget : $Var^+ \times D \mapsto D$ |
| project : $Var^+ \times D \mapsto D$ |
| *Fixpoint iterator operations* |
| isBottom : $D \mapsto \mathbb{B}$ (emptiness test) |
| $\sqsubseteq : D \times D \mapsto \mathbb{B}$ (inclusion test) |
| $\sqcup : D \times D \mapsto D$ (join) |
| $\sqcap : D \times D \mapsto D$ (meet) |
| $\nabla : D \times D \mapsto D$ (widening) |
| $\Delta : D \times D \mapsto D$ (narrowing) |

**Figure 8.** Simplified Abstract Domain API.

*Congruences* [52] expresses constraints of the form $a\mathbb{Z} + b$ where $a$ and $b$ are integers. For instance, all even (odd) numbers are represented as $2\mathbb{Z} + 0$ ($2\mathbb{Z} + 1$). Both domains are combined via a reduced product as described in [52].

These *non-relational* domains are represented by environments from variables to abstract values. CRAB uses *functional maps* [89] to implement efficiently these environments as in [14, 16, 38].

*Zones (a.k.a Difference-Bounds Matrices) [84]* expresses constraints of the form $x - y \leq k$, where $x, y$ are integer variables and $k \in \mathbb{Z}$. CRAB uses an efficient sparse implementation of difference-bounds matrices that achieves sparsity by dynamically separating interval constraints from constraints that can only be expressed through differences [45]. In our experience, Zones is one of our most important domains because it can compute non-trivial relationships between variables while still being efficient in practice (see e.g., [48]).

*Flat Boolean Domain* is a finite lattice $\bot \leq \mathtt{T} \leq \top$, $\bot \leq \mathtt{F} \leq \top$ that discovers which boolean variables are definitely true T or false F. This domain is always combined with a numerical domain so that information can flow between integer to boolean variables.

*DisInt [38]* is an extension of Intervals to a finite disjunction. Elements in this domain are normalized sequences of non-overlapping, sorted intervals $[a_o, b_0], \ldots, [a_n, b_n]$ such that only $a_0$ can be $-\infty$ and $b_n$ can be $+\infty$. This domain retains the scalability of the Interval domain while being able to reason about simple disequalities. For instance, the disequality $\neq 0$ can be expressed by the sequence of intervals $[-\infty, -1]$ and $[1, +\infty]$.

*Boxes [55]* expresses finite boolean combinations of Intervals. Boxes provides an efficient implementation of the exact disjunction of Intervals based on Linear Decision Diagrams [17]. This domain is very useful for path-sensitive analyses. This domain reasons simultaneously about both boolean and integer variables producing much more precise results than the Flat Boolean Domain combined with, for instance, Intervals.

*Numerical domain with uninterpreted functions.* The *Terms* domain [44] can improve the precision of a numerical domain by inferring equivalence amongst sub-expressions based on the theory of uninterpreted functions. Terms is strictly more precise than [87], and it can enhance numerical domains in different ways by: (a) providing some relational information (equalities) to domains such as Intervals and Congruences, (b) improving precision in presence of non-linear operations (e.g., $x \leq 10 \wedge y = \sqrt{x} + y^2 \wedge z = \sqrt{x} + y^2 \to x \leq 10 \wedge y = z$), and (c) improving precision in presence of array operations (e.g., $b = write(a, i, x, sz) \wedge y = select(a, i, sz) \to x = y$).

*Intervals over machine arithmetic.* Most CRAB numerical domains reason about unbounded integers. This forces us to check for integer overflow in order to produce sound verification results. The exception is the Wrapped Interval Domain [88] which infers interval constraints obeying the laws of machine arithmetic, and thus, it can produce correct intervals in the presence of integer wraparounds.

*Interface to Apron and Elina domains.* CRAB provides interfaces to external abstract domains libraries such as Apron [69] and Elina [95]. Thus, domains such as Octagons [86] ($\pm x \pm y \leq k$, where $x, y$ are integer variables and $k \in \mathbb{Z}$) and Polyhedra [30] (linear inequalities of the form $\sum_i c_i \cdot x_i \leq k_i$ where $x_i$ are integer variables, $c_i, k_i \in \mathbb{Z}$) are also available.

*Nullity domain* is a finite lattice $\bot \leq \mathtt{N} \leq \top$, $\bot \leq \mathtt{NN} \leq \top$ that discovers which pointer variables are definitely null N or non-null NN.

*Array content domains* lift abstract domains for scalar variables to reason about arrays. CRAB provides several array content domains that strike different balances between precision and cost. *Array Smashing* [14] treats the whole array as a single symbolic variable. The transfer function for **array_write** can only weaken the previous abstract state (*weak update*). Array Smashing can represent universally quantified invariants if they hold uniformly for all array elements.

On the other extreme, *Array Expansion* [14] treats individually each array element as a single scalar variable. This domain can be more precise because the transfer function for **array_write** can overwrite the old value (*strong update*) and can reason about byte aliasing. However, it might not scale if arrays are too large, and it cannot express universally quantified properties.

Similar to ASTRÉE, CRAB also implements the combination of both domains via a reduced product. Arrays are initially populated using strong updates by the Array Expansion domain. If the size of the array is greater than certain threshold or arrays are accessed using non-constant indexes then they are smashed. A smashed array can be expanded again as in [14]. In CRAB, once an array is smashed it is not expanded again.

CRAB also provides an array content domain [41] based on *Array Partitioning* [49, 65]. The domain is useful when an invariant does not hold for all array elements but instead on some contiguous segment. The domain selects a small set of partition variables, maintaining disjunctive information about properties which hold over the segments delimited by the partition variables. This domain can express properties such as array sortedness but it can suffer from scalability issues. More efficient array domains such as [29] can be also implemented in CRAB.

*Combination of domains.* Most of the abstract domains described above are typically not enough to prove non-trivial properties. However, their combination can produce very powerful analyses [26]. The main method for combining numerical abstract domains in CRAB is the *reduced product* [25]. Given two domains $D_1$ and $D_2$, the reduced product is equivalent to running simultaneously both domains while communicating information between the two domains. This communication is called *reduction*. CRAB provides a generic reduced product domain that redirects each abstract operation to each sub-domain, performing a simple reduction: $\perp$ if any of the sub-domains is $\perp$. More complex reductions are carefully implemented on a case-by-case basis (e.g., Intervals and Congruences, Terms and Zones, Array Smashing and Expansion, etc).

### *4.2.5. Backward Analysis*

When an invariant is too weak to prove an assertion, we can propagate backwards the predecessors of the error states and use the abstract states at those points to prove that there is no an execution starting from the entry of the program that can reach an error state. This approach is good at handling some disjunctive invariants which many of CRAB abstract domains cannot represent precisely. A typical scenario is when an assertion after a join point might not be provable due to loss of precision at the join.

The CRAB backward analysis is based on computing necessary preconditions. A *necessary precondition* of a set of states $F$ is the set of initial states that guarantee that some of its executions will stay in $F$. Similar to [92], CRAB computes in the abstract necessary preconditions starting from the set of error states. These error states are obtained by representing in the abstract the negation of an assertion condition. If the set of initial states is empty then the set of error states must be unreachable, and thus, the assertion definitely holds.

As described in [27], the precision of the backward analysis can be improved by considering only preconditions that might be reachable from the entry of the program. For any basic block $b$, CRAB intersects in the abstract ($\sqcap$ operator) the preconditions from the error states at $b$ with the invariants that hold at $b$. Next, a new forward analysis is run starting from the new preconditions computed by the backward analysis in an attempt to produce more precise invariants. This interleaving process between a forward and backward analysis gives an infinite descending chain of approximated preconditions and invariants whose termination is ensured by narrowing.

### *4.2.6. Inter-Procedural Analysis*

The architecture shown in Figure 6 is limited to intra-procedural analysis. However, inter-procedural analysis is also available in CRAB. Based on our experience, different programs are more amenable to different inter-procedural analyses. To support this view, CRAB is also parametric on the inter-procedural analysis. For a new inter-procedural analysis only these two main steps need to be implemented in CRAB:

- Define the inter-procedural semantics for **call** and **return**.
- Find an effective ordering to traverse the call graph while computing globally stable solutions. Depending on the program, these are the common cases that might need to be considered:
  (1) Incomplete call graph and recursive functions
  (2) Incomplete call graph and non-recursive functions

(3) Complete call graph and recursive functions
(4) Complete call graph and non-recursive functions

Cases (1) and (2) require running simultaneously a pointer analysis together with the abstract domain chosen to reason about the desired property in order to resolve indirect calls. Case (3) can be solved by computing a global fixpoint for each strongly connected component in the call graph. Weak topological ordering [15] can be used to identify widening points. Finally, (4) is the simplest case because the call graph is a directed acyclic graph (DAG).

CRAB implements an inter-procedural analysis based on CGS [101]. The inter-procedural analysis makes the main assumption that the call graph is complete and thus, all the function calls have been already resolved by the client (Section 4.2.7 described how we can ensure that for LLVM programs). A cycle in the call graph is treated by analyzing all the functions in the cycle in an intra-procedural manner. Therefore, the call graph analyzed by CRAB can be considered in practical terms as a DAG. The analysis performs a *context-insensitive*, *summary-based* inter-procedural analysis consisting of two phases:

- Bottom-up (callees before callers): traverse the call graph in reverse topological order while computing summaries for each function. *Summaries* are abstract states relating input with output function parameters. Summaries are computed by first inferring invariants for the function using the intra-procedural analysis and then by producing the actual summary during the transfer function of **return**. While computing invariants, the analysis might need to reuse other summaries from callees. This is implemented in the transfer function of **call**.
- Top-down (callers before callees): traverse the call graph in topological order starting from main. At each call, the inter-procedural semantics for **call** reuses the summary of the callee after formal/actual parameters renaming, as done already during the bottom-up traversal. Moreover, it stores the preconditions associated to that call in the callee. Note that at the time a function is analyzed, all its callers have been already analyzed, and thus, all the preconditions are available. CRAB is context-insensitive because it joins all the preconditions.

The inter-procedural analysis is quite fast since each function is analyzed exactly once. However, the analysis can be imprecise since it is context-insensitive. A context-sensitive analysis can be implemented by keeping separate the preconditions of each function call and then running different analyses starting from each precondition.

### *4.2.7. Clam*

CLAM (CRAB for Llvm Abstraction Manager) is an abstract interpreter for LLVM based on CRAB. The main tasks performed by CLAM are:

1. Translating each LLVM function to a CRAB goto-based Control-Flow Graph.
2. Running CRAB analyses.
3. Assertion checking and/or communicating CRAB invariants to other SeaHorn back-end solvers.

CLAM allows users to choose among several parameters such as the abstract domain, fixpoint parameters (widening delay, number of thresholds, etc.), and whether backward or inter-procedural analysis should be enabled or not. SEAHORN users can choose CLAM as the only back-end engine to discharge proof obligations. However, even if the abstract domain can express precisely the program semantics, due to the join and widening operations, it might lose some precision during the verification. As a consequence, CLAM alone might not be sufficient as a back-end engine. Instead, a more suitable job for CLAM is to supply program invariants to the other engines (e.g. SPACER). For this, the integration between CLAM and SPACER has been carefully tuned. CLAM invariants can be used by SPACER in two different ways. First, invariants can be added as permanent lemmas (i.e., initial may summaries) to initialize each frame $F_i$. In this case, the exploration done by SPACER is limited to states that satisfy the invariants discovered by CLAM. Second, CLAM invariants can be added only to restrict the transition relation during the **Induction** rule. This mode does not interfere with exploration, but can improve generalization done by the rule.

The translation of integer instructions is straightforward. Most of LLVM instructions with integer operands have their CRAB counterparts with the exceptions of $\phi$ and branch instructions which are replaced with CRAB assignments and **assume**, respectively. Similarly, CLAM can translate directly LLVM pointer instructions to CRAB pointer instructions (**alloc**, **load**, and **store**). Note that this syntactic-guided translation approach is pretty simple, but it relies entirely on CRAB to reason about both LLVM registers and memory.

Alternatively, CLAM can perform much of the memory reasoning at translation time by leveraging SEADSA, which is already used during the generation of Constrained Horn Clauses. CLAM can use SEADSA to *disambiguate memory*, i.e., resolve pointer aliasing, which allows us to:

1. Resolve all indirect calls, producing a complete call graph.
2. Perform function *purification*, eliminating all function side-effects.
3. Translate LLVM `load` and `store` instructions to CRAB array instructions **array_select** and **array_write**, respectively.

Steps 1 and 2 greatly simplify CRAB inter-procedural analysis. Step 3 allows leveraging powerful CRAB array domains to infer rich invariants about integer memory values, without complex abstract domains that would need also to reason about pointer aliasing. In our experience, this relatively simple approach has been quite effective at reasoning about C programs.

## 5. Conclusions

We have presented SEAHORN, a software verification framework with a modular design that separates the concerns of the syntax of the language, its operational semantics, and the verification semantics. SEAHORN builds upon two verification engines: SPACER and CRAB. Both SPACER and CRAB represent the state-of-the-art in SMT-based Model Checking and Abstract Interpretation, respectively.

We believe that SEAHORN is a versatile and highly customizable framework that can help significantly in the time-consuming process of building new tools by allowing researchers experimenting only on their particular techniques of interest.

The flexibility and practicality of SEAHORN have been demonstrated by ourselves and other researchers over several projects. In our seminal work [58], we use SEAHORN to prove proper API usage of Linux device drivers and memory safety of autopilot code. In [99], SEAHORN is extended to go beyond safety properties for proving termination of programs. In [70], SEAHORN is used to find code inconsistencies, code fragments without normal terminating executions. In [71], SEAHORN is used to prove safety of smart contracts. In this case, our Clang-based front-end was replaced with an in-house developed front-end that translates Solidity smart contracts directly to LLVM bitcode. In our most recent work, we have used SEAHORN to prove equivalence of x86 executable programs [36]. We use McSema [98] to translate x86 code to LLVM bitcode and then use the SEAHORN Bounded Model Checking engine to prove equivalence of a program and an equivalent variant that is more resilient to cyber-security attacks.

While SEAHORN is already a full featured verification engine, significant work remains to improve both usability and scalability. From the usability perspective, the main questions are around user communication with the tool. Currently, each new property requires non-trivial encoding of a problem domain to a low-level language. Adding support for a new property is non-trivial research-driven effort. On the other side, the results from SEAHORN are difficult to interpret by an average developer. Recently, we have proposed that in the case of counterexamples, the tool must produce an executable that a developer can examine [47]. Note that this is quite different than producing failing inputs, since the executable contains not just the inputs to the original program, but also an executable model of the whole verification environment. While this is a good first step, more work remains to make this practical and robust in the presence of complex programming features including procedures and dynamic memory allocation. From the scalability perspective, dealing with dynamic memory allocation and multi-threaded code are currently the weakest links. We hope to address both by a more modular (but possibly incomplete) reasoning techniques further combining Model Checking and Abstract Interpretation.

## 6. Acknowledgments

SEAHORN would have not been possible without numerous great collaborators. First of all, we would like to thank Temesghen Kashai for being one of the original developers of SEAHORN and main contributor for several years. We would like also to thank (alphabetical order) Nikolaj Bjørner, Graeme Gange, Jeff Gennari, Anvesh Komuravelli, Jakub Kuderski, Peter Schachte, Edward Schwartz, Harald Søndergaard, and Peter Stuckey.

## References

[1] A. Albarghouthi, A. Gurfinkel, and M. Chechik. Craig Interpretation. In *SAS*, pages 300–316, 2012.
[2] A. Albarghouthi, A. Gurfinkel, and M. Chechik. Whale: An interpolation-based algorithm for inter-procedural verification. In *VMCAI*, pages 39–55, 2012.
[3] A. Albarghouthi, Y. Li, A. Gurfinkel, and M. Chechik. Ufo: A framework for abstraction- and interpolation-based software verification. In *CAV*, pages 672–678, 2012.

[4] G. Amato and F. Scozzari. Localizing widening and narrowing. In *SAS*, pages 25–42, 2013.

[5] S. Arlt, C. Rubio-González, P. Rümmer, M. Schäf, and N. Shankar. The gradual verifier. In *NFM*, pages 313–327, 2014.

[6] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software model checking via large-block encoding. In *FMCAD*, pages 25–32, 2009.

[7] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker blast. *STTT*, 9(5-6):505–525, 2007.

[8] D. Beyer and M. E. Keremoglu. Cpachecker: A tool for configurable software verification. In *CAV*, pages 184–190, 2011.

[9] N. Bjørner and A. Gurfinkel. Property directed polyhedral abstraction. In *VMCAI*, pages 263–281, 2015.

[10] N. Bjørner, A. Gurfinkel, K. L. McMillan, and A. Rybalchenko. Horn clause solvers for program verification. In *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, pages 24–51, 2015.

[11] N. Bjørner and M. Janota. Playing with quantified satisfaction. In *LPAR*, pages 15–27, 2015.

[12] N. Bjørner, K. L. McMillan, and A. Rybalchenko. Program verification as satisfiability modulo theories. In *SMT*, pages 3–11, 2012.

[13] N. Bjørner, K. L. McMillan, and A. Rybalchenko. On solving universally quantified horn clauses. In *SAS*, pages 105–125, 2013.

[14] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, pages 196–207, 2003.

[15] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *In Proceedings of the International Conference on Formal Methods in Programming and their Applications*, pages 128–141. Springer-Verlag, 1993.

[16] G. Brat, J. A. Navas, N. Shi, and A. Venet. IKOS: A framework for static analysis based on abstract interpretation. In *SEFM*, pages 271–277, 2014.

[17] S. Chaki, A. Gurfinkel, and O. Strichman. Decision diagrams for linear arithmetic. In *FMCAD*, pages 53–60, 2009.

[18] S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamaric. A reachability predicate for analyzing low-level software. In *TACAS*, pages 19–33, 2007.

[19] A. Cimatti, A. Griggio, and R. Sebastiani. Efficient generation of craig interpolants in satisfiability modulo theories. *ACM Trans. Comput. Log.*, 12(1):7:1–7:54, 2010.

[20] E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In *TACAS*, pages 168–176, 2004.

[21] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. Vcc: A practical system for verifying concurrent c. In *TPHOL*, pages 23–42, 2009.

[22] L. Cordeiro, B. Fischer, and J. Marques-Silva. Smt-based bounded model checking for embedded ANSI-C software. *IEEE Trans. Software Eng.*, 38(4):957–974, 2012.

[23] P. Cousot. Abstracting induction by extrapolation and interpolation. In *VMCAI*, pages 19–42, 2015.

[24] P. Cousot and R. Cousot. Static Determination of Dynamic Properties of Programs. In *Proceedings of the second international symposium on Programming, Paris, France*, pages 106–130, 1976.

[25] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[26] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, 1979.

[27] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *J. Log. Program.*, 13(2&3):103–179, 1992.

[28] P. Cousot and R. Cousot. Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In *PLILP*, pages 269–295, 1992.

[29] P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *POPL*, pages 105–118. ACM, 2011.

[30] P. Cousot and N. Halbwachs. Automatic discovery of linear constraints among variables of a program. In *POPL*, pages 84–97. ACM, 1978.

[31] Crab: A language-agnostic library for Abstract Interpretation. Available from `https://github.com/seahorn/crab`.

[32] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c: A software

analysis perspective. In *SEFM*, pages 233–247, 2012.
[33] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. VeriMAP: A tool for verifying programs through transformations. In *TACAS*, pages 568–574, 2014.
[34] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.
[35] R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.
[36] B. Dutertre, I. Mason, and J. A. Navas. Proving equivalence of x86 programs with McSema and SeaHorn, 2018. Blog available at `http://seahorn.github.io/seahorn/mcsema/equivalence/x86/binary/2018/12/12/seahorn-and-mcsema.1.html`.
[37] Ethereum. The Ethereum EVM JIT. Available at `https://github.com/ethereum/evmjit`.
[38] M. Fähndrich and F. Logozzo. Static contract checking with abstract interpretation. In *FoVeOOS*, pages 10–30, 2010.
[39] G. Fedyukovich, A. Gurfinkel, and A. Gupta. Lazy but effective functional synthesis. In *VMCAI*, pages 92–113, 2019.
[40] G. Fedyukovich, A. Gurfinkel, and N. Sharygina. Property directed equivalence via abstract simulation. In *CAV*, pages 433–453, 2016.
[41] G. Gange, J. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. A partial-order approach to array content analysis. Technical report, `https://arxiv.org/pdf/1408.1754.pdf`, 2014.
[42] G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. Failure tabled constraint logic programming by interpolation. *TPLP*, 13(4-5):593–607, 2013.
[43] G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. Horn clauses as an intermediate representation for program analysis and transformation. *TPLP*, 15(4-5):526–542, 2015.
[44] G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. An abstract domain of uninterpreted functions. In B. Jobstmann and K. R. M. Leino, editors, *VMCAI*, pages 85–103, 2016.
[45] G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. Exploiting sparsity in difference-bound matrices. In *SAS*, pages 189–211, 2016.
[46] P. Garoche, T. Kahsai, and C. Tinelli. Incremental invariant generation using logic-based automatic abstract transformers. In *NASA NFM*, pages 139–154, 2013.
[47] J. Gennari, A. Gurfinkel, T. Kahsai, J. A. Navas, and E. J. Schwartz. Executable counterexamples in software model checking. In *VSTTE*, pages 17–37, 2018.
[48] E. Gershuni, N. Amit, A. Gurfinkel, N. Narodytska, J. A. Navas, N. Rinetzky, L. Ryzhyk, and M. Sagiv. Simple and precise static analysis of untrusted linux kernel extensions. In *PLDI*, 2019.
[49] D. Gopan, T. Reps, and M. Sagiv. A framework for numeric analysis of array operations. In *POPL*, pages 338–350. ACM, 2005.
[50] D. Gopan and T. W. Reps. Lookahead widening. In *CAV*, pages 452–466, 2006.
[51] D. Gopan and T. W. Reps. Guided static analysis. In *SAS*, pages 349–365, 2007.
[52] P. Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 30:165–190, 1989.
[53] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, pages 405–416, 2012.
[54] A. Griggio, T. T. H. Le, and R. Sebastiani. Efficient interpolant generation in satisfiability modulo linear integer arithmetic. *Logical Methods in Computer Science*, 8(3), 2010.
[55] A. Gurfinkel and S. Chaki. Boxes: A symbolic abstract domain of boxes. In *SAS*, pages 287–303, 2010.
[56] A. Gurfinkel and S. Chaki. Combining predicate and numeric abstraction for software model checking. *STTT*, 12(6):409–427, 2010.
[57] A. Gurfinkel, S. Chaki, and S. Sapra. Efficient Predicate Abstraction of Program Summaries. In *NFM*, pages 131–145, 2011.
[58] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The SeaHorn Verification Framework. In *CAV*, pages 343–361, 2015.
[59] A. Gurfinkel, T. Kahsai, and J. A. Navas. SeaHorn: A framework for verifying C programs - (competition contribution). In *TACAS*, 2015.
[60] A. Gurfinkel and J. A. Navas. A context-sensitive memory model for verification of C/C++ programs. In *SAS*, pages 148–168, 2017.
[61] A. Gurfinkel, S. Shoham, and Y. Vizel. Quantifiers on demand. In *ATVA*, pages 248–266, 2018.
[62] A. Gurfinkel and Y. Vizel. DRUPing for interpolates. In *FMCAD*, pages 99–106, 2014.

[63] A. Gurfinkel, O. Wei, and M. Chechik. Model checking recursive programs with exact predicate abstraction. In *ATVA*, pages 95–110, 2008.

[64] N. Halbwachs and J. Henry. When the decreasing sequence fails. In *SAS*, pages 198–213, 2012.

[65] N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *PLDI*, pages 339–348. ACM, 2008.

[66] M. Heizmann, J. Christ, D. Dietsch, E. Ermis, J. Hoenicke, M. Lindenmann, A. Nutz, C. Schilling, and A. Podelski. Ultimate Automizer with SMTInterpol - (Competition Contribution). In *TACAS*, pages 641–643, 2013.

[67] K. Hoder and N. Bjørner. Generalized property directed reachability. In *SAT*, pages 157–171, 2012.

[68] J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa. TRACER: A symbolic execution tool for verification. In *CAV*, pages 758–766, 2012.

[69] B. Jeannet and A. Miné. A library of numerical abstract domains for static analysis. In A. Bouajjani and O. Maler, editors, *CAV*, volume 5643 of *LNCS*, pages 661–667. Springer, 2009.

[70] T. Kahsai, J. A. Navas, D. Jovanovic, and M. Schäf. Finding inconsistencies in programs with loops. In *LPAR*, pages 499–514, 2015.

[71] S. Kalra, S. Goel, M. Dhawan, and S. Sharma. ZEUS: analyzing safety of smart contracts. In *NDSS*, 2018.

[72] A. Katis, G. Fedyukovich, H. Guo, A. Gacek, J. Backes, A. Gurfinkel, and M. W. Whalen. Validity-guided synthesis of reactive systems from assume-guarantee contracts. In *TACAS*, pages 176–193, 2018.

[73] A. Komuravelli, N. Bjørner, A. Gurfinkel, and K. L. McMillan. Compositional verification of procedural programs using horn clauses over integers and arrays. In *FMCAD*, pages 89–96, 2015.

[74] A. Komuravelli, A. Gurfinkel, and S. Chaki. SMT-based model checking for recursive programs. In *CAV*, pages 17–34, 2014.

[75] A. Komuravelli, A. Gurfinkel, and S. Chaki. Smt-based model checking for recursive programs. *Formal Methods in System Design*, 48(3):175–205, 2016.

[76] L. Lakhdar-Chaouch, B. Jeannet, and A. Girault. Widening with thresholds for programs with complex control graphs. In *ATVA*, pages 492–502, 2011.

[77] A. Lal and S. Qadeer. A program transformation for faster goal-directed search. In *FMCAD*, pages 147–154, 2014.

[78] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88, 2004.

[79] C. Lattner and V. S. Adve. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In *PLDI*, pages 129–142, 2005.

[80] K. McMillan and A. Rybalchenko. Solving Constrained Horn Clauses using Interpolation. Technical report, MSR-TR-2013-6, 2013.

[81] M. Méndez-Lojo, J. A. Navas, and M. V. Hermenegildo. A flexible, (C)LP-based approach to the analysis of object-oriented programs. In *LOPSTR*, pages 154–168, 2007.

[82] F. Merz, S. Falke, and C. Sinz. LLBMC: bounded model checking of C and C++ programs using a compiler IR. In *VSTTE*, pages 146–161, 2012.

[83] B. Mihaila, A. Sepp, and A. Simon. Widening as abstract domain. In *NASA NFM*, pages 170–184, 2013.

[84] A. Miné. A few graph-based relational numerical abstract domains. In *SAS*, pages 117–132, 2002.

[85] A. Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *LCTES*, pages 54–63, 2006.

[86] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[87] A. Miné. Symbolic methods to enhance the precision of numerical abstract domains. In *VMCAI*, pages 348–363, 2006.

[88] J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. Signedness-agnostic program analysis: Precise integer bounds for low-level code. In R. Jhala and A. Igarashi, editors, *APLAS*, volume 7705 of *LNCS*, pages 115–130. Springer, 2012.

[89] C. Okasaki and A. Gill. Fast mergeable integer maps. In *Notes of the ACM SIGPLAN Workshop on ML*, pages 77–86, September 1998.

[90] J. C. Peralta, J. P. Gallagher, and H. Saglam. Analysis of imperative programs through analysis of constraint logic programs. In *SAS*, pages 246–261, 1998.

[91] Z. Rakamaric and M. Emmi. SMACK: Decoupling source language details from verifier implementations. In *CAV*, pages 106–113, 2014.
[92] X. Rival. Understanding the origin of alarms in astrée. In *SAS*, pages 303–319, 2005.
[93] P. Rümmer, H. Hojjat, and V. Kuncak. Disjunctive interpolants for horn-clause verification. In *CAV*, pages 347–363, 2013.
[94] A. Simon and A. King. Widening polyhedra with landmarks. In *APLAS*, pages 166–182, 2006.
[95] G. Singh, M. Püschel, and M. T. Vechev. ELINA: ETH Library for Numerical Analysis, 2018. Available at `https://github.com/eth-sri/ELINA`.
[96] N. Sinha, N. Singhania, S. Chandra, and M. Sridharan. Alternate and learn: Finding witnesses without looking all over. In *CAV*, pages 599–615, 2012.
[97] B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, pages 32–41, 1996.
[98] TrailOfBits. Framework for lifting x86, amd64, and aarch64 program binaries to llvm bitcode. Available at `https://github.com/trailofbits/mcsema`.
[99] C. Urban, A. Gurfinkel, and T. Kahsai. Synthesizing ranking functions from bits and pieces. In *TACAS*, pages 54–70, 2016.
[100] A. Venet. A scalable nonuniform pointer analysis for embedded programs. In *SAS*, pages 149–164, 2004.
[101] A. Venet and G. P. Brat. Precise and efficient static array bound checking for large embedded C programs. In *PLDI*, pages 231–242, 2004.
[102] J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *POPL*, pages 427–440, 2012.

This page intentionally left blank

*Engineering Secure and Dependable Software Systems*
*A. Pretschner et al. (Eds.)*
*IOS Press, 2019*
*© 2019 The authors and IOS Press. All rights reserved.*
*doi:10.3233/978-1-61499-977-5-113*

# Using Epistemic Logic to Analyze Protocols

Joseph Y. Halpern [1]
*Computer Science Department, Cornell University, Ithaca, NY 14850, USA*
*halpern@cs.cornell.edu*

**Keywords.** epistemic logic, coodinated attack, blockchain, authentication logic, BAN logic, common knowledge

I gave a sequence of four lectures. The first two were largely taken from the paper "Knowledge and common knowledge in a distributed environment" (Halpern and Moses 1990). By first considering a number of puzzles and paradoxes, I argued that the right way to understand distributed protocols is by considering how messages change the state of knowledge of a system. I presented a hierarchy of knowledge states that a system may be in, and discussed how communication can move the system's state of knowledge up the hierarchy. Of special interest is the notion of common knowledge. Common knowledge is an essential state of knowledge for reaching agreements and coordinating action. I considered the *coordinated attack problem* (Gray 1978), which showed that in practical distributed systems, common knowledge is not attainable. I then introduced various relaxations of common knowledge that are attainable in many cases of interest. I showed that common knowledge was necessary and sufficient for simultaneous coordination; the relaxations all correspond to relaxations of coordination. For example, $\epsilon$-common knowledge (roughly speaking, within $\epsilon$ everyone will know that within $\epsilon$ everyone will know that within $\epsilon$ ...) is necessary and sufficient for coordination within a window of size $\epsilon$ and probabilistic coordination (with high probability everyone knows that with high probability everyone knows ...) is necessary and sufficient for probabilistic coordination. The book *Reasoning About Knowledge* (Fagin, Halpern, Moses, and Vardi 1995) goes into much more detail about the use of epistemic logic for analyzing protocols, as well as providing logical details, such as soundness and completeness proofs for various epistemic logics.

My third and fourth lectures considered recent applications of epistemic logic. The third one was entitled "A knowledge-based analysis of the blockchain protocol", and based on joint work with Rafael Pass (2017). As the title suggests, it focused on the *blockchain* protocol, a protocol for achieving consensus on a public ledger that records bitcoin transactions. To the extent that a blockchain protocol is used for applications such as contract signing and making certain transactions (such as house sales) public, we need to understand what guarantees the protocol

[1]Thanks to NSF, AFOSR, and ARO for funding over the years. Currently supported by NSF grants IIS-1703846 and IIS-1718108, AFOSR grant FA9550-12-1-0040, ARO grant W911NF-17-1-0592, and the Open Philanthropy project.

gives us in terms of agents' knowledge. I provided a complete characterization of agents' knowledge when running a blockchain protocol, using a variant of common knowledge that takes into account (1) the fact that agents can enter and leave the system, (2) that it is not known which agents are in fact following the protocol (some agents may want to deviate if they can gain by doing so), and (3) the fact that the guarantees provided by blockchain protocols are probabilistic. The key was using an appropriate variant of common knowledge that, roughly speaking, says that within $\Delta$ time units (where $\Delta$ is an upper bound on message delivery time in the system), from that time onward, all the honest players will know that within $\Delta$ times units, from that time onward, all the honest players will know .... Note that the set of honest players is an *indexical set*; its membership changes over time as players enter and leave the system, and can be different in different runs (histories) of the system. Thus, making sense of this notion of common knowledge requires considering common knowledge relative to indexical sets, a topic first considered by Moses and Tuttle (1988). It also requires *agent-relative knowledge* (so that we can say "I know", rather than "agent $i$ knows"), a topic first considered by Grove and Halpern (Grove 1995; Grove and Halpern 1993).

The last lecture was entitled "An epistemic foundation for authentication logics", and based on joint work with Ron van der Meyden and Riccardo Pucella (2017). While there have been many attempts, going back to BAN logic, to base reasoning about security protocols on epistemic notions, they have not been all that successful. Arguably, this has been due to the particular logics chosen. I presented a simple logic based on the well-understood modal operators of knowledge, time, and probability, and showed that it is able to handle issues that have often been swept under the rug by other approaches, while being flexible enough to capture all the higher-level security notions that appear in BAN logic. Moreover, while still assuming that the knowledge operator allows for unbounded computation, it can handle the fact that a computationally bounded agent cannot decrypt messages in a natural way, by distinguishing strings and message terms. These become different types, with translations between them, so that we can say, for example, that a string $s$ represents a particular message $m$. However, $s$ may represent $s$ in one run and not another. This approach allows us to say, for example, (1) that agent $i$ knows that $s$ represents the encryption of a particular message $m$; (2) that $i$ knows that $s$ represents the encryption of some message, even though $i$ does not know which message it is the encryption of; (3) that $i$ knows that encryptions are unique; and (4) that $i$ knows that $s$ represents the encryption of a message of length at most 20. I showed that this logic can capture BAN logic notions by providing a translation of the BAN operators into our logic, capturing belief by a form of probabilistic knowledge.

## References

Fagin, R., J. Y. Halpern, Y. Moses, and M. Y. Vardi (1995). *Reasoning About Knowledge.* Cambridge, MA: MIT Press. A slightly revised paperback version was published in 2003.

Gray, J. (1978). Notes on database operating systems. In R. Bayer, R. M. Graham, and G. Seegmuller (Eds.), *Operating Systems: An Advanced Course*, Lecture Notes in Computer Science, Volume 66. Berlin/New York: Springer-Verlag. Also appears as IBM Research Report RJ 2188, 1978.

Grove, A. J. (1995). Naming and identity in epistemic logic II: a first-order logic for naming. *Artificial Intelligence 74*(2), 311–350.

Grove, A. J. and J. Y. Halpern (1993). Naming and identity in epistemic logics, Part I: the propositional case. *Journal of Logic and Computation 3*(4), 345–378.

Halpern, J. Y., R. v. d. Meyden, and R. Puccella (2017). An epistemic foundation for authenticatino logics. In *Theoretical Aspects of Rationality and Knowledge: Proc. Sixteenth Conference (TARK 2017)*, pp. 306–323. The proceedings are published in *Electronic Proceedings in Theoretical Computer Science* **251**.

Halpern, J. Y. and Y. Moses (1990). Knowledge and common knowledge in a distributed environment. *Journal of the ACM 37*(3), 549–587. A preliminary version appeared in *Proc. 3rd ACM Symposium on Principles of Distributed Computing*, 1984.

Halpern, J. Y. and R. Pass (2017). A knowledge-based analysis of the blockchain protocol. In *Theoretical Aspects of Rationality and Knowledge: Proc. Sixteenth Conference (TARK 2017)*, pp. 324–335. The proceedings are published in *Electronic Proceedings in Theoretical Computer Science* **251**.

Moses, Y. and M. R. Tuttle (1988). Programming simultaneous actions using common knowledge. *Algorithmica 3*, 121–169.

This page intentionally left blank

*Engineering Secure and Dependable Software Systems*
*A. Pretschner et al. (Eds.)*
*IOS Press, 2019*
*© 2019 The authors and IOS Press. All rights reserved.*
*doi:10.3233/978-1-61499-977-5-117*

# Abstraction-Based Control Design (Lecture Notes)

Rupak MAJUMDAR, Kaushik MALLIK, and Anne-Kathrin SCHMUCK
*Max Planck Institute for Software Systems, Germany*

**Abstract** Abstraction-based control design (ABCD) is a methodology to algorithmically construct controllers for continuous time dynamical systems for temporal specifications. The ABCD algorithm constructs a finite and discrete-time abstraction of the dynamical system, computes a discrete controller for the finite system using reactive synthesis, and refines the discrete controller to a controller for the original dynamical system. These lecture notes lay out the basic theory for ABCD and describe a recent extension which performs the abstraction in a multi-layered, adaptive way.

**Keywords.** Abstraction-based control design, reactive synthesis

## 1. Introduction

Control of dynamical systems concerns the feedback design of trajectories that satisfy a given set of specifications. Classically, the primary specifications were stability (boundedness of trajectories), regulation (maintaining an nominal behavior), and set-point tracking (tracking a reference signal). In recent years, though, many systems design problems come with more complex specifications about the temporal behavior of the trajectory. Moreover, classical techniques were only well understood for simpler linear dynamical systems, and would often rely on human judgment in the controller design process: tuning a PID controller is a classic example. In contrast, the modern automatic control systems have become seemingly more complicated with large state spaces and highly non-linear dynamics. Modern applications (like aviation) also often require strong correctness guarantees for the control system, which hand-tuning would fail to provide.

This has led to a flurry of new, *formally correct* and *algorithmic* techniques for control design for non-linear and hybrid systems. One particularly fruitful direction is the interaction between the "classical control" techniques for control of continuous systems and the automata-theoretic techniques arising out of the control of discrete systems.

The key to this interaction is abstraction: a way to discretize the continuous dynamics so that one can apply automata-theoretic techniques on the discretization while maintaining a refinement back to the original continuous system. The goal of *abstraction-based controller design (ABCD)* is to automatically synthesize controllers for non-linear dynamical systems such that the closed-loop system satisfies a temporal logic specification. In this approach, a time-sampled version of the continuous dynamics of the open-loop system (the *concrete system*) is abstracted by a symbolic finite state model (the *abstract system*). Then, automata-theoretic algorithms from finite-state reactive synthesis are used to synthesize a discrete controller on the abstract system for a given temporal logic specification. If there is a *feedback refinement relation* (FRR) between the concrete and abstract systems, the abstract controller can be refined to a controller for the concrete system such that the resulting closed loop satisfies the specification.

Usually, the abstract system is computed by first fixing a parameter $\tau$ for the sample time and a parameter $\eta$ for the state and input spaces, and then representing the abstract state space as a set of hypercubes, each of diameter $\eta$. The hypercubes partition the continuous concrete state space. The abstract transition relation adds a transition between two hypercubes iff there exists some state in the first hypercube which can reach some state of the second by following the original dynamics for time $\tau$. This results in a transition system that over-approximates the effect of the original dynamics on the abstract state space.

The goal of these lecture notes is to provide a basic understanding of ABCD. For readability, we focus only on temporal reachability and safety properties; the techniques can be extended to all $\omega$-regular properties through usual automata-theoretic means. Reachability means that a given target set is eventually reached. Safety means that a given set of safe states is never left.

The basic ideas of ABCD are, by now, well understood and supported by several robust tools—`SCOTS`, `MASCOT`, or `PFACES`, to name a few. The key bottleneck to these methods is the size of the abstraction, which grows exponentially with the system dimension. Tackling this state-space explosion is one of the main open directions of research.

The success of ABCD depends on the choice of $\eta$ and $\tau$. Intuitively, increasing $\eta$ (and $\tau$)[1] results in fewer hypercubes but leads to a more imprecise abstract transition relation. Thus, one may not be able to find a controller for the abstract system. On the other hand, decreasing $\eta$ (and $\tau$) results in a more precise abstraction and a higher chance of successful controller synthesis. However, the larger state space can make the synthesis problem computationally intractable. This observation has led to an extension of ABCD to a *multi-layered* setting, where the algorithm maintains several "layers" of abstract systems by picking hypercube partitions of different resolution. The resulting abstract controller synthesis procedure tries to find a controller for the coarsest abstraction whenever feasible, but adaptively considers finer abstractions when necessary. We also describe a recent multi-layered ABCD algorithm for reachability and safety.

The multi-layered algorithms explained in these notes, and their lazy extension has been implemented in a tool called `MASCOT`.[2] On a number of examples, `MASCOT` demonstrates the advantage of using multi-layered techniques in ABCD.

The rest of the notes are organized as follows. We start by defining the setting of ABCD (Section 2) and then describe synthesis algorithms for reachability and safety properties in the multi-layered setting (Sections 3 and 4). We conclude with some pointers to the research literature.

## 2. Preliminaries

**Notation.** We use the symbols $\mathbb{N}$, $\mathbb{R}$, $\mathbb{R}_{\geq 0}$, $\mathbb{R}_{>0}$, $\mathbb{Z}$, and $\mathbb{Z}_{>0}$ to denote the sets of natural numbers, reals, non-negative reals, positive reals, integers, and positive integers, re-

[1]Usually, $\tau$ is increased along with $\eta$ to reduce non-determinism due to self loops.

[2]`http://mascot.mpi-sws.org`

spectively. Given $a,b \in \mathbb{R}$ s.t. $a \leq b$, we denote by $[a,b]$ a closed interval and define $[a;b] = [a,b] \cap \mathbb{Z}$ as its discrete counterpart. Given $a,b \in \mathbb{R}^n$, we denote by $a_i$ and $b_i$ their $i$-th element and write $[\![a,b]\!]$ for the closed hyper-interval $\mathbb{R}^n \cap ([a_1,b_1] \times \ldots \times [a_n,b_n])$. We define the relations $<, \leq, \geq, >$ on $a,b$ component-wise.

For a set $W$, we write $W^*$, $W^+$, and $W^\omega$ for the sets of finite sequences, non-empty finite sequences, and infinite sequences over $W$, respectively. We define $W^\infty = W^* \cup W^\omega$. For $w \in W^*$, we write $|w|$ for the length of $w$; the length of $w \in W^\omega$ is $\infty$. We define $\mathrm{dom}(w) = \{0,\ldots,|w|-1\}$ if $w \in W^*$, and $\mathrm{dom}(w) = \mathbb{N}$ if $w \in W^\omega$. We denote by $\mathrm{dom}^+(w) = \mathrm{dom}(w) \setminus \{0\}$ the positive domain of $w$. For $k \in \mathrm{dom}(w)$ we write $w(k)$ for the $k$-th symbol of $w$ and $w|_{[0,k]}$ for the restriction of $w$ to the domain $[0,k]$. If $W = A \times B$, the projection of $w \in W^\infty$ on $A$ is denoted by $w|_A$.

Given two sets $A$ and $B$, $f : A \rightrightarrows B$ and $f : A \rightarrow B$ denote a set-valued and ordinary map, respectively. The map $f$ is called *strict* if $f(a) \neq \emptyset$ for all $a \in A$. We identify set-valued maps with their respective binary relation over $A \times B$, i.e., $(a,b) \in f$ iff $b \in f(a)$. The inverse mapping $f^{-1} : B \rightrightarrows A$ is defined via its respective binary relation: $f^{-1}(b) = \{a \in A \mid b \in f(a)\}$.

## 2.1. Abstraction-Based Controller Synthesis

We now recall the general procedure of abstraction-based controller synthesis (ABCD) using the framework of feedback refinement relations (FRR) as introduced in [RWR17].

**Systems.** A *system* $S = (X,U,Y,F,H)$ consists of a state space $X$, an input space $U$, an output space $Y$, and set-valued maps $F : X \times U \rightrightarrows X$ and $H : X \times U \rightrightarrows Y$ representing the transition function and the output function, respectively. A system $S$ is *finite* if $X$, $U$, and $Y$ are finite. It is *simple* if $X = Y$ and $H(x,u) = x$ for all $x \in X$ and $u \in U$, and *static* if $X$ is a singleton. If $S$ is simple (resp. static) we use the triple $S = (X,U,F)$ (resp. $S = (U,Y,H)$ with $H : U \rightrightarrows Y$) for notational convenience. The *behavior* $\mathscr{B}(S)$ of a system $S = (X,U,Y,F,H)$ is given by the set

$$\{\xi \in X^\infty \mid \forall k \in \mathrm{dom}^+(\xi) \,.\, \xi(k) \in \textstyle\bigcup_{u \in U} F(\xi(k-1),u)\}. \tag{1}$$

**Controllers and Closed Loop Systems.** Given a simple system $S = (X,U,F)$, a *controller* (to be precise, a state-feedback controller) $C$ of $S$ is a function $C : X \rightarrow U$. Given a system $S = (X,U,F)$ and a controller $C$, the *closed loop system* formed by interconnecting $S$ and $C$ in *feedback* is defined by the system $S^{cl} = (X,U,F^{cl})$ where[3]

$$F^{cl}(x,u) = \left\{ x' \,\middle|\, \begin{array}{l} u \in C(x) \wedge \\ x' \in F(x,u) \end{array} \right\}. \tag{2}$$

**Safety and Reachability Control Problems.** We consider *safety* and *reachability* properties as control objectives. Below, we use $\Box$ and $\Diamond$ to denote the temporal operators "always" and "eventually". Given a set $T$ of states of $S$, we write $\Box T$ for a safety objective and define $\langle\!\![\Box T]\!\!\rangle_S \subseteq \mathscr{B}(S)$ as the set of behaviors in $\mathscr{B}(S)$ which always remain within the set $T$:

[3]In contrast to the definition used in [RWR17], we keep the input used in $F^{cl}$ explicit. This allows us to apply feedback refinement relations to closed loop systems.

$$\{\xi\in X^\infty \mid \forall k\in\mathrm{dom}^+(\xi).\xi(k)\in T\}$$

We write $\Diamond T$ for a reachability objective and define $\langle\!\![\Box T]\!\!\rangle_S \subseteq \mathscr{B}(S)$ as the set of behaviors in $\mathscr{B}(S)$ which eventually reach the set $T$:

$$\{\xi\in X^\infty \mid \exists k\in\mathrm{dom}^+(\xi).\xi(k)\in T\}$$

Given a system $S$ and a safety or reachability objective $\psi$, the pair $\langle S,\psi\rangle$ is called a *control problem* on $S$ for $\psi$. A controller $C$ of $S$ solves the control problem $\langle S,\psi\rangle$ if $\mathscr{B}(S^{cl})|_X \subseteq \langle\!\![\psi]\!\!\rangle_S$. The set of all controllers solving $\langle S,\psi\rangle$ is denoted by $\mathscr{C}(S,\psi)$.

**Feedback Refinement Relations.** Let $S_i=(X_i,U_i,F_i)$, $i\in\{1,2\}$ be two systems, and suppose $U_2\subseteq U_1$. A *feedback refinement relation* (FRR) from $S_1$ to $S_2$ is a strict relation $Q\subseteq X_1\times X_2$ s.t. for all $(x_1,x_2)\in Q$, we have (i) $U_{S_2}(x_2)\subseteq U_{S_1}(x_1)$, and (ii) $u\in U_{S_2}(x_2)\Rightarrow Q(F_1(x_1,u))\subseteq F_2(x_2,u)$ where $U_{S_i}(x):=\{u\in U_i \mid F_i(x,u)\neq\emptyset\}$. We write $S_1\preccurlyeq_Q S_2$ if $Q$ is an FRR from $S_1$ to $S_2$.

Consider two simple systems $S_1$ and $S_2$, with $S_1\preccurlyeq_Q S_2$. Let $C$ be a controller of $S_2$. Then $C$ can be refined into a controller for $S_1$ given as $C\circ Q$, where "$\circ$" is the usual function composition. As shown in [RWR17], the refinement is sound.

**Proposition 1 ([RWR17], Def. VI.2, Thm. VI.3)** *Let $S_1\preccurlyeq_Q S_2$ and $C\in\mathscr{C}(S_2,\psi)$ for a specification $\psi$. If for all $\xi_1\in\mathscr{B}(S_1)$ and $\xi_2\in\mathscr{B}(S_2)$ with* $\mathrm{dom}(\xi_1)=\mathrm{dom}(\xi_2)$ *and $(\xi_1(k),\xi_2(k))\in Q$ for all $k\in$* $\mathrm{dom}(\xi_1)$ *it holds that $\xi_2\in\langle\!\![\psi]\!\!\rangle_{S_2}\Rightarrow\xi_1\in\langle\!\![\psi]\!\!\rangle_{S_1}$, then $C\circ Q\in\mathscr{C}(S_1,\psi)$.*

## *2.2. ABCD for Continuous Control Systems*

We now recall how ABCD can be applied to continuous-time systems by delineating the abstraction procedure [RWR17].

**Continuous-Time Control Systems.** A *control system* $\Sigma=(X,U,W,f)$ consists of a state space $X=\mathbb{R}^n$, a non-empty input space $U\subseteq\mathbb{R}^m$, a disturbance space $W=[\![-w,w]\!]$ s.t. $w\in\mathbb{R}^n_{\geq 0}$ and a nonlinear differential inclusion

$$\dot{\xi}\in f(\xi(t),u(t))+W, \tag{3}$$

where $f(\cdot,u)$ fulfills the usual conditions for existence and uniqueness of solution of the differential equation $\dot{\xi}=f(\xi(t),u(t))$. (For example, a sufficient condition for existence and uniqueness is that $f(\cdot,u)$ is locally Lipschitz continuous for all $u\in U$.) $\Sigma$ defines a perturbed continuous-time nonlinear system, and $w$ is a component-wise bound on perturbations to its dynamics.m,

Given a positive parameter $\tau>0$ and a constant input trajectory $\mu_u:[0,\tau]\to U$ which maps every $t\in[0,\tau]$ to $u$, a solution of the inclusion in (3) on $[0,\tau]$ is an absolutely continuous function $\xi:[0,\tau]\to X$ that fulfills (3) for almost every $t\in[0,\tau]$. We collect all such solutions in the set $\mathrm{Sol}_f(\tau,u)$. Given an initial condition $x_0\in X$, the solution to the unperturbed control system $\dot{\xi}=f(\xi(t),u(t))$ associated with (3) is unique, and its value at time $t\in[0,\tau]$ is denoted by $\zeta(t,x_0,\mu)$. Given a subset of states $X'\subseteq X$, and a subset of inputs $U'\subseteq U$ s.t. $[0,\tau]\times X'\times U'\subseteq\mathrm{dom}(\zeta)$, the map $\beta_\tau:\mathbb{R}^n_{\geq 0}\times U'\to\mathbb{R}^n_{\geq 0}$ is a *growth bound* on $X'$ and $U'$ associated with $\tau$ and (3) if

$$\begin{aligned}
&\forall r, r' \in \mathbb{R}^n_{\geq 0}, u \in U' \;.\; r \geq r' \Rightarrow \beta_\tau(r,u) \geq \beta_\tau(r',u) \;\text{ and}\\
&\forall\, \xi \in \mathrm{Sol}_f(\tau,u), x_0 \in X' \;.\\
&\quad \xi(0) \in X' \Rightarrow |\xi(\tau) - \zeta(\tau, x_0, \mu)| \leq \beta_\tau(|\xi(0) - x_0|, u).
\end{aligned}$$

**Time-Sampled System.** Given a time sampling parameter $\tau > 0$, we define by $\vec{S}(\Sigma,\tau) = (X, U, \vec{F})$ the (simple) *time-sampled system* associated with $\Sigma$, where

$$x' \in \vec{F}(x,u) \Leftrightarrow \exists \xi \in \mathrm{Sol}_f(\tau,u) \;.\; \xi(0) = x \wedge \xi(\tau) = x'. \tag{4}$$

The state space of $\vec{S}(\Sigma,\tau)$ is still infinite; we next define a finite system associated with $\Sigma$.

**Abstract System.** A *cover* $\hat{X}$ of the state space $X$ is a set of non-empty, closed hyper-intervals $[\![a,b]\!]$ with $a, b \in (\mathbb{R} \cup \{\pm\infty\})^n$ called *cells*, such that every $x \in X$ belongs to some cell in $\hat{X}$. We assume that there exists a compact subset $X' \subseteq X$ of the state space, which is quantized by compact cells, whereas the (unbounded) region covered by $\hat{X} \setminus \hat{X}'$ is not of interest to the control problem and is covered by a finite number of large unbounded cells.

Given a *grid parameter* $\eta \in \mathbb{R}^n_{>0}$ and $X' \subseteq X$ with $X' = [\![\alpha, \beta]\!]$ s.t. $\beta - \alpha = k\eta$ for some $k \in \mathbb{Z}^n$, the set

$$\eta\mathbb{Z}^n = \{c \in X' \mid \exists k \in \mathbb{Z}^n . (\forall i \in [1;n]. c_i = \alpha_i + k_i\eta_i - 0.5 * \eta_i)\} \tag{5}$$

defines the center points of cells in $\hat{X}'$ with diameter $\eta$, i.e.

$$\hat{x} \in \hat{X}' \Rightarrow \exists c \in \eta\mathbb{Z}^n \;.\; \hat{x} = c + [\![-\eta/2, \eta/2]\!]. \tag{6}$$

This results in congruent cells which are uniformly aligned on a grid.[4] We denote by $c_{\hat{x}}$ the unique center point of $\hat{x}$.

The (simple) system $\hat{S}(\Sigma, \tau, \eta, \beta_\tau) = (\hat{X}, \hat{U}, \hat{F})$ is a *symbolic abstract system* associated with $\Sigma$, $\tau$, $\eta$, and $\beta_\tau$ if the following holds: (i) $\hat{X}$ is a finite cover of $X$, there exists a non-empty subset $\hat{X}' \subseteq \hat{X}$ s.t. $\hat{X}'$ satisfies (6), and $\beta_\tau$ is a growth bound[5] on $\hat{X}'$ and $\hat{U}$, (ii) a finite $\hat{U} \subseteq U$, (iii) for all $\hat{x} \in \hat{X} \setminus \hat{X}'$ and $u \in \hat{U}$, $\hat{F}(\hat{x},u) = \emptyset$, and (iv) for all $\hat{x} \in \hat{X}'$, $\hat{x}' \in \hat{X}$, and $u \in \hat{U}$, $\hat{x}' \in \hat{F}(\hat{x},u)$ iff

$$\left(\zeta(\tau, c_{\hat{x}}, u) + [\![-\beta_\tau(\frac{\eta}{2}, u), \beta_\tau(\frac{\eta}{2}, u)]\!]\right) \cap \hat{x}' \neq \emptyset. \tag{7}$$

The computation of $\hat{F}(\hat{x},u)$ has been illustrated in Fig. 1. If the control system $\Sigma$ and parameters $\tau$, $\eta$, and $\beta_\tau$ are clear from the context, we omit them in $\vec{S}$ and $\hat{S}$.

[4]In the standard distribution of SCOTS, the grid is aligned such that (an extension of) $\eta\mathbb{Z}^n$ has a center point that coincides with the origin. Shifting this grid by $\eta/2$ yields our grid alignment.

[5]We consider a universal growth bound for notational simplicity. There exist systems where abstract controller synthesis is successful only if different growth bounds for different regions of the state space are considered (see [WRR16]).

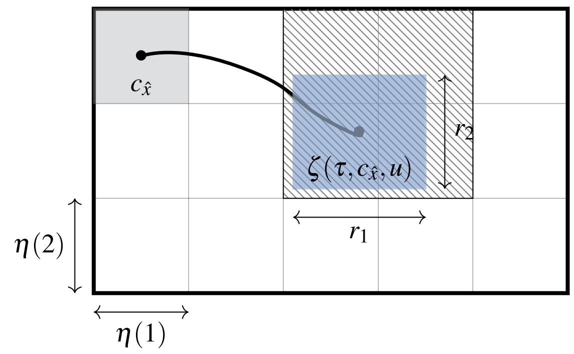


**Figure 1.** Computation of abstract transitions based on growth bound: the gray box is the abstract state $\hat{x}$, the blue box is the region $\left(\zeta(\tau, c_{\hat{x}}, u) + [\![-\beta_\tau(\frac{\eta}{2}, u), \beta_\tau(\frac{\eta}{2}, u)]\!]\right)$, where we used the notation $r_1, r_2$ to denote the first and the second component of $2\beta_\tau(\frac{\eta}{2}, u)$, and the 4 cells covered with hatch-lines constitute the set $\hat{F}(\hat{x}, u)$.

**Induced FRR.** It was shown in [RWR17], Thm. III.5 that the relation $\hat{Q} \subseteq X \times \hat{X}$ defined by $(x, \hat{x}) \in \hat{Q}$ iff $x \in \hat{x}$ is an FRR between $\vec{S}$ and $\hat{S}$, i.e., $\vec{S} \preccurlyeq_{\hat{Q}} \hat{S}$. Hence, we can apply ABCD as described in Sec. 2.1 by computing a controller for $\hat{S}$ which can then be refined to a controller for $\vec{S}$ under the pre-conditions of Prop. 1.

## 2.3. *Algorithms for Safety and Reachability*

We use the following notational conventions from $\mu$-calculus [BS06] and linear temporal logic [MP92]. Let $f$ denote a monotone operator on a finite set $Q$, i.e., $f(P') \subseteq f(P'') \subseteq Q$ for all $P' \subseteq P'' \subseteq Q$. The least and greatest fixed-points exist uniquely and are denoted $\mu P.f(P)$ and $\nu P.f(P)$, respectively.

Given a system $S$, we define a function on sets of states:

$$\mathrm{Pre}_S(A) = \{\hat{x} \mid \exists \hat{u} \,.\, \hat{F}_l(\hat{x}, \hat{u}) \subseteq A_l\}$$

for any $A \subseteq \hat{x}$.

The reachability control problem $(S, \Diamond T)$ can be solved by computing the minimal fixed-point [MPS95]:

$$\mathscr{W}[\Diamond T] = \mu W \,.\, \mathrm{Pre}_S(W) \cup T. \tag{8}$$

This fixed-point is evaluated iteratively by computing

$$W^0 = T \text{ and } W^{i+1} = \mathrm{Pre}_S(W^i) \cup T \tag{9}$$

until we reach some $N \in \mathbb{N}$ s.t. $W^N = W^{N+1}$. For a finite system, the iterations are guaranteed to terminate.

Similarly, safety controllers can be synthesized by evaluating the maximal fixed-point [MPS95]

$$\mathscr{W}[\Box T] = \nu V \,.\, \mathrm{Pre}_S(V) \cap T \tag{10}$$

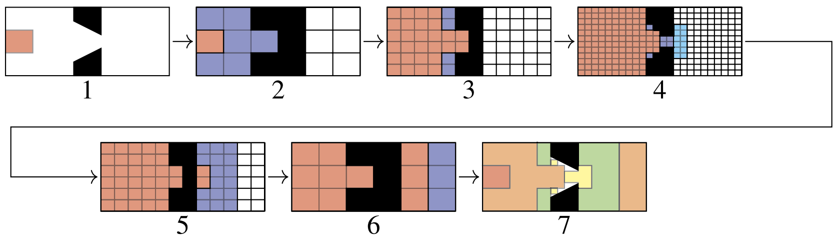


**Figure 2.** Application of the algorithm in Fig. 3 to the reachability problem in Pic. 1 using $m = 2$ and three layers $l = 1$ (Pic. 4), $l = 2$ (Pic. 3, 5) and $l = 3$ (Pic. 2, 6). In Pics. 1-6, the target ($T$) and the obstacles are indicated in red and black, respectively, and the winning states ($W$) are indicated in blue and cyan depending on whether they were computed in the first attempt or the second attempt, respectively. Pic. 7 indicates the domains of the resulting controllers with different granularity: $l = 1$ (yellow), $l = 2$ (green), and $l = 3$ (orange).

by computing

$$V^0 = \hat{x} \text{ and } V^{i+1} = T \cap \mathrm{Pre}_S(V^i) \tag{11}$$

iteratively until we reach a fixed point.

## 3. Multi-Layered ABCD

We now extend ABCD to a multi-layered setting with $L$ layers. Our exposition follows [HMMS18c,HMMS18b]. The abstract states in layer $l \in [1;L]$ are hyper-cells with parameter $\eta_l$, and the transitions are discrete jumps with sampling time $\tau_l$. A control system $\Sigma$ may be abstracted *non-uniformly* by using different layers for different parts of the state space.

### *3.1. Informal Overview*

We first provide an informal overview of multi-layered ABCD.

Consider the problem of controlling the motion of a vehicle in a given space (Pic. 1 in Fig. 2) s.t. a specified target (red region) is reached while avoiding static obstacles (black regions). Fix $\eta$ and $\tau$, and consider three abstractions $S_1$, $S_2$, and $S_3$ of the system, with parameters $(\eta, \tau)$, $(2\eta, 2\tau)$, and $(4\eta, 4\tau)$, respectively. Thus, $S_1$ is the finest abstraction, and $S_3$ is the coarsest. Each abstract state space is a set of squares (2D hypercubes) partitioning the underlying 2D space. The abstract transition function for each $S_i$ allows transitions only between cells that share a corner.

Given Fig. 2, we see that we can only cross the passage between the obstacles by using $S_1$, yet $S_2$ and $S_3$ suffice in the remaining parts of the state space. This suggests an algorithm, like the one informally stated in Fig. 3, which tries to use the coarsest abstraction $S_3$, but switches to a finer abstraction (and back) when necessary.

When applying this algorithm to the problem in Pic. 1 of Fig. 2, we start by running a usual fixed-point algorithm for reachability (REACH) on $S_3$ using the red states as the target. All states returned by REACH (marked in blue in Pic. 2) allow us to apply a (sequence of) control input(s) to steer the system to the red region, and are therefore usually called *winning states*.

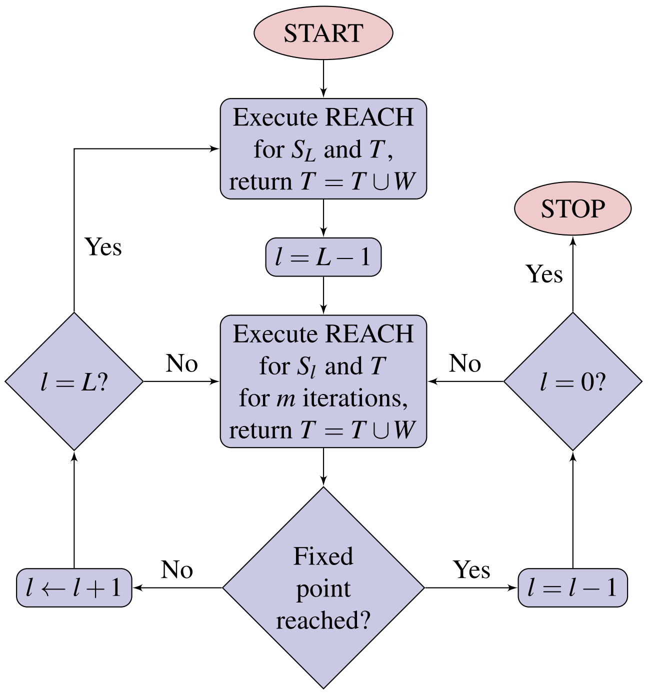

START
Execute REACH for $S_L$ and $T$, return $T = T \cup W$
$l = L-1$
Execute REACH for $S_l$ and $T$ for $m$ iterations, return $T = T \cup W$
Fixed point reached?
No
$l \leftarrow l+1$
$l = L$?
Yes
No
Yes
$l = l-1$
$l = 0$?
No
Yes
STOP

**Figure 3.** Flow chart of the proposed multi-layered reachability fixed-point. $T$ and $W$ denote the target and the winning states, respectively. There are $L$ layers and $m$ is a parameter determining when to switch to a more abstract layer.

At this point, we proceed to the next finer layer $S_2$ and run REACH for $S_2$ by using the previously computed winning states as the new target (marked in red in Pic. 3). We see that this fixed-point terminates after one step. Therefore, we go down to $S_1$ (Pic. 4).

At this point, we could run the controller synthesis algorithm in $S_1$ until convergence. While correct, this does not capture the intuition that $S_2$ and $S_3$ suffice for controller synthesis after crossing the passage. Hence, our synthesis algorithm uses a tuning parameter $m$ (equal to 2 in Fig. 2) to specify that, for layers that are not the coarsest, the reachability fixed-point iterates for up to $m$ steps and returns the current winning set. After iterating REACH on $S_1$ for $m = 2$ steps (gaining the blue states in Pic. 4), we switch to layer 2 and see that no new states are added to the target set for $S_2$ compared to Pic. 3. We thus immediately return to $S_1$ and run REACH for two more iterations (gaining the cyan states in Pic. 4). As REACH did not converge for $S_1$, we move to $S_2$ and run $m = 2$ iterations of REACH over $S_2$, but now for the new target (Pic. 5). Since this also does not converge, we finally move back to $S_3$ and run REACH until convergence (Pic. 6). At this point, we can verify that the set of states is a fixed-point for every layer; hence, the algorithm terminates.

As a by-product of the algorithm, we obtain a partition of the state space where each sub-region corresponds to the domain of one locally computed controller. This is illustrated in Pic. 7 of Fig. 2. The overall abstract controller is given by the union of all local ones, and its soundness follows from the fact that each local controller treats the union of all lower ranked controllers' domains (i.e., controllers computed earlier in the algorithm) as its target. By refining this controller to the underlying concrete system, we obtain a controller with non-uniform resolution which works as follows: for each continuous state $x$ whose related abstract state $\hat{x}$ is in the domain of one local controller computed for layer $l$, the control input corresponding to $\hat{x}$ is applied for duration $\tau_l$ to the underlying control system.

## 3.2. Multi-Layered Systems

Given a grid parameter $\underline{\eta}$, a time sampling parameter $\underline{\tau}$, and $L \in \mathbb{Z}_{>0}$, define[6] $\eta_l = 2^{l-1}\underline{\eta}$ and $\tau_l = 2^{l-1}\underline{\tau}$. Fix a control system $\Sigma$ and a subset $X' \subseteq X$ with $X' = [\![\alpha, \beta]\!]$, s.t. $\beta - \alpha = k\eta_L$ for some $k \in \mathbb{Z}^n$. For each $l \in [1;L]$, we define the grid $\eta_l\mathbb{Z}^n$ and the cover $\hat{X}_l'$ as in (5) and (6), respectively, by substituting $\eta$ with $\eta_l$. This construction induces a sequence of time-sampled systems $\{\vec{S}_l(\Sigma, \tau_l)\}_{l\in[1;L]}$ and a sequence of symbolic abstract systems $\{\hat{S}_l(\Sigma, \tau_l, \eta_l, \beta_{\tau_l})\}_{l\in[1;L]}$. For simplicity, we assume that all layers use the same continuous and abstract input spaces $U$ and $\hat{U} \subseteq U$, respectively. If $\Sigma$, $\tau_l$, $\eta_l$, and $\beta_{\tau_l}$ are clear from the context, we use $\vec{S}_l$ and $\hat{S}_l$ as short forms of $\vec{S}_l(\Sigma, \tau_l)$ and $\hat{S}_l(\Sigma, \tau_l, \eta_l, \beta_{\tau_l})$, respectively.

It trivially follows from our construction that, for all $l \in [1;L]$, we have $\vec{S}_l \preccurlyeq_{\hat{Q}_l} \hat{S}_l$, where $\hat{Q}_l \subseteq X \times \hat{X}_l$ is the FRR induced by $\hat{X}_l$. The set of relations $\{\hat{Q}_l\}_{l\in[1;L]}$ induces transformers $\hat{R}_{ll'} \subseteq \hat{X}_l \times \hat{X}_{l'}$ for $l, l' \in [1;L]$ between abstract states of different layers such that

$$\hat{x} \in \hat{R}_{ll'}(\hat{x}') \Leftrightarrow \hat{x} \in \hat{Q}_l(\hat{Q}_{l'}^{-1}(\hat{x}')). \tag{12}$$

Given the sequence $\{\hat{S}_l\}_{l\in[1;L]}$ with $\hat{S}_l = (\hat{X}_l, \hat{U}, \hat{F}_l)$, we can use $\hat{R}_{ll'}$ to define the (simple) *abstract multi-layered system* $\hat{\mathbf{S}} = (\hat{\mathbf{X}}, \hat{U}, \hat{\mathbf{F}})$, where $\hat{\mathbf{X}} = \bigcup_{l\in[1;L]} \hat{X}_l$ and

$$\hat{\mathbf{F}}(\hat{x}, \hat{u}) = \bigcup_{l\in[1;L]} \hat{R}_{ll'}(\hat{F}_{l'}(\hat{x}, \hat{u})) \tag{13}$$

for all $\hat{x} \in \hat{X}_{l'}$, $l' \in [1;L]$, and $\hat{u} \in \hat{U}$. Intuitively, $\hat{\mathbf{S}}$ is a non-deterministic system; for every state $\hat{x} \in \hat{X}_{l'}$ and input $\hat{u} \in \hat{U}$, transitions to states of all layers are possible. That is, $\hat{x}' \in \hat{\mathbf{F}}(\hat{x}, \hat{u})$ if there exists an $\hat{x}'' \in \hat{X}_{l'}$ s.t. $\hat{x}'' \in \hat{F}_{l'}(\hat{x}, \hat{u})$ and $\hat{x}' \in \hat{R}_{ll'}(\hat{x}'')$ for some $l \in [1;L]$.

Similarly, given the sequence $\{\vec{S}_l\}_{l\in[1;L]}$ with $\vec{S}_l = (X_l, U, \vec{F}_l)$, we define a (simple) *time-sampled multi-layered system* $\vec{\mathbf{S}} = (X, U, \vec{\mathbf{F}})$ s.t.

$$\vec{\mathbf{F}}(x, u) = \bigcup_{l\in[1;L]} \vec{F}_l(x, u) \tag{14}$$

for all $x \in X$ and $u \in U$. Again, $\vec{\mathbf{S}}$ is a non-deterministic system; in every state $x \in X$ transitions of any duration $\tau_l$, $l \in [1;L]$ can be chosen, which correspond to some $\vec{F}_l(x, u)$.

The behaviors $\mathscr{B}(\hat{\mathbf{S}})$ and $\mathscr{B}(\vec{\mathbf{S}})$ are defined via (1).

[6] Our method is applicable to grid parameters defined by $\eta_1 = \underline{\eta}$ and $\eta_{l+1} = \gamma\eta_l$ for $l \in [1;L-1]$ and $\gamma \in \{2^k \mid k \in \mathbb{N}\}$, and sampling times $\tau_l = \alpha(l)\underline{\tau}$ where $\alpha : [1;L] \to \mathbb{R}$ is a monotonically increasing function. For notational simplicity, we restrict our attention to $\gamma = 2$ and $\alpha(l) = 2^{l-1}$.

**Remark 1** *Even though we have $\vec{S}_l \preccurlyeq_{\hat{Q}_l} \hat{S}_l$ for all $l \in [1;L]$, each relation is evaluated for a different sampling time $\tau_l$. Therefore, the relations $\hat{R}_{ll'}$ cannot define a FRR between $\hat{S}_l$ and $\hat{S}_{l'}$ using the definition from Sec. 2.*

*Given that $\tau_l = 2^{l-1}\underline{\tau}$, a natural extension of FRR seems to be that, for any $(\hat{x}_{l+1}, \hat{x}_l) \in \hat{R}_{(l+1)l}$, it holds that*

$$\hat{u} \in \hat{U} \Rightarrow \hat{R}_{(l+1)l}(\hat{F}_l(\hat{F}_l(\hat{x}_l, \hat{u}), \hat{u})) \subseteq \hat{F}_{l+1}(\hat{x}_{l+1}, \hat{u}). \tag{15}$$

*That is, we would expect that states $\hat{x}_l \in \hat{X}_l$ and $\hat{x}_{l+1} \in \hat{X}_{l+1}$ related via $\hat{R}_{(l+1)l}$ remain related when the $l$-th layer transition function is applied to $\hat{x}_l$ twice for $\tau_l$ (resulting in a duration $2\tau_l = \tau_{l+1}$) and when the $l+1$-th layer transition function is only applied once to $\hat{x}_{l+1}$ (also resulting in a duration $\tau_{l+1}$). While this seems intuitive, we can prove that* (15) *does not hold in general. This is because applying the $l$-th layer transition function twice introduces an additional over-approximation step which is caused by* (7).

## *3.3. Multi-Layered Controllers*

Given $\hat{\mathbf{S}}$ as in (13) and some $P \in \mathbb{N}$, we define a *multi-layered controller* $\mathbf{C}$ for $\hat{\mathbf{S}}$ as $\mathbf{C} = \{C_p\}_{p \in [1;P]}$ s.t.

$$\forall p \in [1;P] \,.\, \exists! l_p \in [1;L] \,.\, \mathrm{dom}(C_p) \subseteq \hat{X}_{l_p}. \tag{16}$$

We do not require any connection between $P$ and $L$. In particular, we allow for layers to have multiple controllers, i.e., $l_p = l_q$ for $p, q \in [1;P]$, $p \neq q$, and no controller at all, i.e. there might be $l \in [1;L]$ s.t. there exists no $p \in [1;P]$ s.t. $l_p = l$.

Given a multi-layered controller $\mathbf{C}$ of $\hat{\mathbf{S}}$, we define the *quantizer induced by* $\mathbf{C}$ as the strict map $\mathbf{Q} : X \rightrightarrows \hat{\mathbf{X}}$ s.t. for all $x \in X$ it holds that $\hat{x} \in \mathbf{Q}(x)$ iff either
(i) there exists a $p \in [1;P]$ s.t.

$$\hat{x} \in \hat{Q}_{l_p}(x) \wedge \hat{x} \in \mathrm{dom}(C_p)$$

and there exists no other $p' \in [1;P]$ with $l_{p'} > l_p$ s.t.

$$\hat{R}_{l_{p'} l_p}(\hat{x}) \in \mathrm{dom}(C_{p'}),$$

or (ii) $\hat{x} \in \hat{Q}_1(x)$ and there exists no $p \in [1;P]$ s.t.

$$\hat{R}_{l_p 1}(\hat{x}) \in \mathrm{dom}(C_p).$$

We define $\mathrm{img}(\mathbf{Q}) = \{\hat{x} \in \hat{\mathbf{X}} \mid \exists x \in X \,.\, \hat{x} \in \mathbf{Q}(x)\}$. Intuitively, $\mathbf{Q}$ maps states $x \in X$ to the coarsest abstract state $\hat{x}$ that is both related to $x$ and in the domain of the controller $\mathbf{C}$ (condition (i)). However, if such an abstract state does not exist, $\mathbf{Q}$ maps $x$ to its related layer $l = 1$ states (condition (ii)). It should be noted that $\hat{Q}_l$ is non-deterministic for states which lie at the boundary of two cells $\hat{x}, \hat{x}' \in \hat{X}_l$. If such an $x$ happens to be located at the boundary of controller domains computed for different layers, $\mathbf{Q}$ maps $x$ to two abstract cells within different layers.

### 3.4. Multi-Layered Closed Loops

Given a multi-layered controller $\mathbf{C}$ of a multi-layered abstract system $\hat{\mathbf{S}}$, the usual construction of a closed loop in (2) results in a system which selects the layer $l'$ of the next state non-deterministically (due to (13)). This will result in a blocking-behavior of the closed loop whenever the selected layer does not correspond to a layer currently admitting a control input.

We therefore propose a different closed-loop definition for multi-layered systems which restricts available transitions to those connecting states in the image of $\mathbf{Q}$. Formally, the closed loop formed by $\hat{\mathbf{S}}$ and $\mathbf{C}$ is defined as the *abstract multi-layered closed-loop system*

$$\begin{aligned} &\hat{\mathbf{S}}^{cl} = (\hat{\mathbf{X}}, \hat{U}, \hat{\mathbf{F}}^{cl}) \text{ s.t.} && (17) \\ &\hat{\mathbf{F}}^{cl}(\hat{x}, \hat{u}) = \left\{ \hat{x}' \middle| \exists p \in [1;P] \,.\, \hat{x} \in \operatorname{img}(\mathbf{Q}) \cap \hat{X}_{l_p} \wedge \hat{u} \in C_p(\hat{x}) \wedge \hat{x}' \in \Lambda_p(\hat{x}, \hat{u}) \right\} \end{aligned}$$

where $\hat{x}' \in \Lambda_p(\hat{x}, \hat{u})$ iff

$$\exists \hat{x}'' \in \hat{F}_{l_p}(\hat{x}, \hat{u}) \,.\, \hat{x}' \in \mathbf{Q}(\hat{Q}_{l_p}^{-1}(\hat{x}'')).$$

When refining $\mathbf{C}$ via $\mathbf{Q}$ to a controller for $\vec{\mathbf{S}}$, the resulting controller should select (i) the current input $u \in \hat{U} \subseteq U$, and (ii) the duration $\tau_l$ for which this input should be applied to the underlying continuous system. As $\mathbf{Q}$ might map a particular state $x \in X$ to multiple abstract states within different layers, we cannot define the refined controller as the serial composition $\mathbf{C} \circ \mathbf{Q}$ in analogy to Sec. 2.1. Instead, we directly define the *time-sampled multi-layered closed loop system* in analogy to $\hat{\mathbf{S}}^{cl}$ by

$$\begin{aligned} &\vec{\mathbf{S}}^{cl} = (X, \hat{U}, \vec{\mathbf{F}}^{cl}) \text{ s.t.} && (18) \\ &\vec{\mathbf{F}}^{cl}(x, \hat{u}) = \left\{ x' \middle| \begin{array}{c} \exists p \in [1;P] \,.\, \hat{x} \in \mathbf{Q}(x) \cap \hat{X}_{l_p} \wedge \\ \hat{u} \in C_p(\hat{x}) \wedge x' \in \vec{F}_{l_p}(x, \hat{u}) \end{array} \right\}. \end{aligned}$$

$\mathscr{B}(\hat{\mathbf{S}}^{cl})$ and $\mathscr{B}(\vec{\mathbf{S}}^{cl})$ can now be defined via (1).

### 3.5. Soundness

Intuitively, $S_1 \preccurlyeq_Q S_2$ ensures that, given a state $x_1 \in X_1$, no matter which related state $x_2 \in Q(x_1)$ and enabled control input $u \in U_{S_2}(x_2)$ is used, all resulting states in $F_1(x_1, u)$ and $F_2(x_2, u)$ are related. This intuition can be transferred to the closed loop systems $\hat{\mathbf{S}}^{cl}$ and $\vec{\mathbf{S}}^{cl}$ as follows. Intuitively, $\hat{\mathbf{S}}^{cl}$ and $\vec{\mathbf{S}}^{cl}$ can generate all closed-loop trajectories without resolving the non-determinism induced by the set-valued maps $\mathbf{Q}$ and $C_p$. If $\mathbf{Q}$ is an FRR from $\vec{\mathbf{S}}^{cl}$ to $\hat{\mathbf{S}}^{cl}$, any implementation resolving this non-determinism in $\mathbf{Q}$ and $C_p$ returns a sound closed loop. This leads us to the following theorem.

**Theorem 1** *Let* $\mathbf{C}$ *be a multi-layered controller for the abstract multi-layered system* $\hat{\mathbf{S}}$*,* $\mathbf{Q}$ *be the quantizer induced by* $\mathbf{C}$*, and* $\vec{\mathbf{S}}^{cl}$ *and* $\hat{\mathbf{S}}^{cl}$ *be the time-sampled and abstract multi-layered closed loop systems defined in* (18) *and* (17)*, respectively. Then* $\vec{\mathbf{S}}^{cl} \preccurlyeq_{\mathbf{Q}} \hat{\mathbf{S}}^{cl}$.

PROOF: We prove both conditions for FRR separately.
(i) Show $\hat{\mathbf{F}}^{cl}(\hat{x},\hat{u}) \neq \emptyset \Rightarrow \vec{\mathbf{F}}^{cl}(x,\hat{u}) \neq \emptyset$: First observe that for any $l \in [1;L]$, it follows from the construction of $\vec{S}_l$ and $\hat{S}_l$ from $\Sigma$ that, for any $\hat{u} \in \hat{U} \subseteq U$, $\hat{x} \in \hat{X}_l$, and $x \in X$, it holds that $\hat{F}_l(\hat{x},\hat{u}) \neq \emptyset$ and $\vec{F}_l(x,\hat{u}) \neq \emptyset$. With this, it follows from (17) and the fact that $\mathbf{Q}$ is strict that $\hat{\mathbf{F}}^{cl}(\hat{x},\hat{u}) \neq \emptyset$ iff $\hat{x} \in \mathrm{img}(\mathbf{Q}) \cap \hat{X}_{l_p}$ and $\hat{u} \in C_p(\hat{x})$, implying $\vec{\mathbf{F}}^{cl}(\hat{x},\hat{u}) \neq \emptyset$ (from (18)).
(ii) Pick $(x,\hat{x}) \in \mathbf{Q}$ and $\hat{u} \in U_{\hat{\mathbf{S}}^{cl}}(\hat{x})$ and show $\mathbf{Q}(\vec{\mathbf{F}}^{cl}(x,\hat{u})) \subseteq \hat{\mathbf{F}}^{cl}(\hat{x},\hat{u})$: first, consider the case that $\hat{x} \in \hat{Q}_1(x)$ and case (ii) in the definition of $\mathbf{Q}$ holds. Then $\hat{\mathbf{F}}^{cl}(\hat{x},\hat{u}) = \emptyset$ and hence $U_{\hat{\mathbf{S}}^{cl}}(\hat{x}) = \emptyset$, i.e., the statement trivially holds. Therefore, assume that case (i) of the definition of $\mathbf{Q}$ holds, implying that there exists some $p \in [1;P]$ s.t. $\hat{x} \in \mathbf{Q}(x) \cap \hat{X}_{l_p}$ and $\hat{x} \in \mathrm{dom}(C_p)$. This implies $\hat{u} \in C_p(\hat{x})$ (from part (i)) and therefore $\hat{x}' \in \mathbf{Q}(\vec{\mathbf{F}}^{cl}(x,\hat{u}))$ iff

$$\hat{x}' \in \mathbf{Q}(\vec{F}_{l_p}(x,\hat{u})). \tag{19}$$

Now recall that $\vec{S}_{l_p} \preccurlyeq_{\hat{Q}_{l_p}} \hat{S}_{l_p}$, hence $\hat{Q}_{l_p}(\vec{F}_{l_p}(x,\hat{u})) \subseteq \hat{F}_{l_p}(\hat{x},\hat{u})$. With this we see that (19) implies $\hat{x}' \in \mathbf{Q}(\hat{Q}_{l_p}^{-1}(\hat{F}_{l_p}(\hat{x},\hat{u})))$ and hence (19) implies $\hat{x}' \in \Lambda_p(\hat{x},\hat{u})$. With this, it immediately follows from (17) that $\hat{x}' \in \hat{\mathbf{F}}^{cl}(\hat{x},\hat{u})$. □

Using the properties of feedback refinement relations, Thm. 1 implies that the usual soundness property of ABCD stated in Prop. 1 can be transferred to the multi-layered setting. This is summarized by the following corollary.

**Corollary 1** *Given the preliminaries of Thm. 1, let* $\mathbf{C} \in \mathscr{C}(\hat{\mathbf{S}},\psi)$ *for a specification* $\psi$ *with associated behavior* $\langle\!\langle\psi\rangle\!\rangle_{\hat{\mathbf{S}}} \subseteq \mathscr{B}(\hat{\mathbf{S}})$ *and* $\langle\!\langle\psi\rangle\!\rangle_{\vec{\mathbf{S}}} \subseteq \mathscr{B}(\vec{\mathbf{S}})$. *Suppose that for all* $\xi \in \mathscr{B}(\vec{\mathbf{S}})$ *and* $\hat{\xi} \in \mathscr{B}(\hat{\mathbf{S}})$ *s.t. (i)* $\mathrm{dom}(\xi) = \mathrm{dom}(\hat{\xi})$, *(ii) for all* $k \in \mathrm{dom}(\xi_1)$, $(\xi(k),\hat{\xi}(k)) \in \mathbf{Q}$, *and (iii)* $\hat{\xi} \in \langle\!\langle\psi\rangle\!\rangle_{\hat{\mathbf{S}}} \Rightarrow \xi \in \langle\!\langle\psi\rangle\!\rangle_{\vec{\mathbf{S}}}$. *Then* $\mathscr{B}(\vec{\mathbf{S}}^{cl})|_X \subseteq \langle\!\langle\psi\rangle\!\rangle_{\vec{\mathbf{S}}}$, *i.e., the time-sampled multi-layered closed loop* $\vec{\mathbf{S}}^{cl}$ *defined in* (18) *fulfills specification* $\psi$.

Cor. 1 can be interpreted as follows. Consider the control system $\Sigma$ at state $x_0$. This state is mapped by $\mathbf{Q}$ to $\hat{x}_0 \in \mathrm{dom}(C_p)$ for some $p$. Choosing any $u_0 \in C_p(\hat{x}_0)$ and applying this input for time $\tau_{l_p}$ to $\Sigma$ results in a continuous trajectory $\xi$ with $x_0 = \xi(0)$ and $x_1 = \xi(\tau_{l_p}) \in \vec{F}_{l_p}(x_0,u)$. Reapplying this procedure leads to an infinite trajectory $\xi$, with sampled version $\vec{\xi} = x_0x_1\ldots \in \mathscr{B}(\vec{\mathbf{S}})$ and abstract version $\hat{\xi} = \hat{x}_0\hat{x}_1\ldots \in \mathscr{B}(\hat{\mathbf{S}})$. As $\mathbf{C} \in \mathscr{C}(\hat{\mathbf{S}},\psi)$ condition (iii) in Cor. 1 ensures that $\vec{\xi} \in \langle\!\langle\psi\rangle\!\rangle_{\vec{\mathbf{S}}}$. For reachability, this implies that $\vec{\xi}$ (and $\xi$) eventually reaches the target. For safety, this implies that $\vec{\xi}$ (i.e. sampling instances of $\xi$) lies inside the safe set.

## 4. Multi-Layered Synthesis

We now provide algorithms for synthesizing multi-layered controllers for given (abstract) control problems which are compatible with $\hat{\mathbf{S}}$.

### 4.1. Reachability

Given an abstract multi-layered system $\hat{\mathbf{S}} = (\hat{\mathbf{X}}, \hat{U}, \hat{\mathbf{F}})$, recall that a *reachability control problem* is the control problem $(\hat{\mathbf{S}}, \Diamond T)$ where $T$ is interpreted as a subset of $\hat{X}_1^l$ and

$$\langle\![\Diamond T]\!\rangle_{\hat{\mathbf{S}}} = \{\hat{\xi} \in \mathscr{B}(\hat{\mathbf{S}}) \mid \exists k \in \mathrm{dom}(\hat{\xi}) \, . \, \hat{\xi}(k) \in T\}. \tag{20}$$

We use the Pre operator to multi-layered systems in the obvious way: $\mathrm{Pre}_{\hat{S}_l}(A_l) = \{\hat{x} \mid \exists \hat{u} \, . \, \hat{F}_l(\hat{x}, \hat{u}) \subseteq A_l\}$ for any $A_l \subseteq \hat{X}_l$ and layer $l$.

One thing to note is that the set $T$ might not be exactly representable by the abstract states in $\hat{X}$. In that case, in order to ensure soundness, we consider the largest underapproximation of the set $T$ for the synthesis, which is defined below:

$$\widehat{T} = \{\hat{x} \in \hat{X} \mid \hat{x} \subseteq T\}. \tag{21}$$

**Single-Layered Systems.** When $L = 1$, the reachability control problem reduces to $(\hat{S}_1, \Diamond T)$, and is solved by computing the minimal fixed-point [MPS95]:

$$\mathscr{W}[\Diamond T] = \mu W \, . \, \mathrm{Pre}_{\hat{S}_1}(W) \cup \widehat{T}. \tag{22}$$

This fixed-point is evaluated iteratively by computing

$$W^0 = \widehat{T} \text{ and } W^{i+1} = \mathrm{Pre}_{\hat{S}_1}(W^i) \cup \widehat{T} \tag{23}$$

until we reach some $N \in \mathbb{N}$ s.t. $W^N = W^{N+1}$. This defines a controller $C$ with $\mathrm{dom}(C) = W^N \setminus \widehat{T}$ and

$$C(x) = \{\hat{u} \in \hat{U} \mid \hat{F}(x, \hat{u}) \subseteq W^{i^*}\} \tag{24}$$

for all $x \in \mathrm{dom}(C)$, where $i^* = \min\{i \mid x \in W^i \setminus \widehat{T}\} - 1$. The controller $C$ is sound: the closed loop system $\hat{S}_1^{cl}$ composed from $C$ and $\hat{S}_1$ satisfies $\mathscr{B}(\hat{S}_1^{cl}) \subseteq \langle\![\Diamond T]\!\rangle_{\hat{S}_1}$ and the system can be steered to the target $T$ from any state $x$ in the controller domain $\mathrm{dom}(C)$.

Using the set $\{W^i\}_{i \in [1;N]}$ computed by (23) one can also define a multi-layered controller $\mathbf{C} = \{C^i\}_{i \in [1;N]}$ by interpreting each iteration in (23) as the computation of a controller $C^i$, with domain $\mathrm{dom}(C_i) = W^i \setminus W^{i-1}$ and $C_i(x) = \{\hat{u} \in \hat{U} \mid \hat{F}_1(x, \hat{u}) \subseteq W^{i-1}\}$. This interpretation of (23), not too useful when $L = 1$, is convenient for the multi-layered setting.

**Multi-Layered Systems.** The simplest way to extend $\mathbf{C}$ to the general case ($L > 1$) is to pick an arbitrary granularity $l_i$ in every iteration of (23). First, we introduce the operator

$$\Gamma_{ll'}(A_{l'}) = \begin{cases} \hat{R}_{ll'}(A_{l'}), & l \leq l' \\ \{\hat{x}_l \in \hat{X}_l \mid \hat{R}_{l'l}(\hat{x}_l) \subseteq A_{l'}\}, & l > l'. \end{cases} \tag{25}$$

which underapproximates a set $A_{l'} \subseteq \hat{X}_{l'}$ by a set $A_l = \Gamma_{ll'}(A_{l'}) \subseteq \hat{X}_l$. Then, for any sequence $\{l_i\}_{i \geq 1}$ with $l_i \in [1;L]$, we compute

$$W^0 = \widehat{T},\ l_0 = 1 \text{ and}$$
$$W^{i+1} = \mathrm{Pre}_{\hat{S}_{l_{i+1}}}(\Gamma_{l_{i+1}l_i}(W^i)) \cup \Gamma_{l_{i+1}l_1}(\widehat{T})$$

iteratively until $W^{i+1} = \Gamma_{l_{i+1}l_i}(W^i)$. When the fixed-point converges, we get a multi-layered controller $\mathbf{C}$ as described before. It is easy to see that $\mathbf{C}$ is sound no matter how $\{l_i\}$ is chosen: each state in the winning set can be controlled using $\mathbf{C}$ to reach $T$. Unfortunately, this simple algorithm is not complete due to two reasons: (i) the iteration can reach a fixed-point at level $l$ even if more states can be added at a lower level, and (ii) lower level winning states can be lost via $\Gamma_{ll'}$, which only gives an exact map for $l < l'$ (recall Pic. 2 and Pic. 3 in Fig. 2).

To fix these problems, we propose REACHIT$_m$ (Alg. 1), which follows the flowchart in Fig. 3. The overall multi-layered reachability algorithm, ML_REACH$_m$ (with tuning parameter $m \in \mathbb{N}$), calls REACHIT$_m(T,l,\emptyset)$ with target $T$, level $l$, and an empty controller, and returns its result. The subroutine REACH$_m$ called in Alg. 1 (line 2 and 12) runs $m$ iterations of (23). Formally, given the input $\Lambda \subseteq \hat{X}_l$ and $l \in [1;L]$, it returns a set $W \subseteq \hat{X}_l$ s.t. $W = W^j$, where $W^j$ is computed via (23) and either $j \leq m$ with $W^j = W^{j-1}$ (then line 15 of Alg. 1 is true) or $j = m$ otherwise. Additionally, REACH$_m$ returns the controller $C$ computed using (24). We extend REACH$_m$ to REACH$_\infty$ in the obvious way, s.t. the returned $W$ is the fixed-point of (23). The set $\Upsilon$ is used in Alg. 1 to save computed winning states to the lowest layer $l = 1$. To see why this is necessary, recall Pic. 4 in Fig. 2. There, we first computed REACH for $l = 1$ and $m = 2$ (obtaining the blue states) and moved to $l = 2$; observe that no winning states were added in $l = 2$. Had we not saved the blue states in $\Upsilon$, they would have been lost and been re-generated after returning to $l = 1$, causing an infinite loop.

**Soundness and Relative Completeness.**[7] To see why ML_REACH$_m$ is sound, consider the following induction on the depth of recursive calls $d$ of REACHIT$_m$. For depth 1 (base case), the single controller that we get in the set $\mathbf{C}$ is sound: this follows from the fact that REACH$_\infty$ (in Alg. 1 Line. 2) is sound. Moreover, it is easy to observe that the controller obtained in depth $d+1$ can enforce a visit (in at most $m$ steps) to the winning region of the controller obtained in depth $d$ (from Line 8, 19, and 23), implying soundness by induction.

Now recall that any controller $C$ computed via REACH has the property that any state in $B$ can reach the target. Therefore, the multi-layered controller $\mathbf{C}$ computed by ML_REACH$_m$ is complete w.r.t. the single-layered controller $C$ computed via REACH if

$$\hat{Q}_1^{-1}(\mathrm{dom}(C)) \subseteq \bigcup_{d\in[1;D]} \mathbf{Q}^{-1}(\mathrm{dom}(C^d)), \tag{26}$$

where $D$ is the maximum number of recursive calls of REACHIT$_m$, and $C^d$ is the controller synthesized in recursion depth $d$ of REACHIT$_m$.

First, it should be noted that for any state $x \in X$ for which $\mathbf{Q}(x) \cap \mathrm{dom}(C^d) \neq \emptyset$ holds, we have: if there exists $d' \in [1;D]$ and $\hat{x} \in \mathbf{Q}(x) \cap \mathrm{dom}(C^d)$ s.t. $\hat{R}_{l_{d'}l_d}(\hat{x}) \cap \mathrm{dom}(C^{d'}) \neq \emptyset$ then $d' \leq d$. Hence, the quantizer $\mathbf{Q}$ formally defined via a ranking over layers $l_d$ in

[7] Absolute completeness of controller synthesis cannot be guaranteed by ABCD; we therefore provide completeness relative to the finest layer.

**Algorithm 1** REACHIT$_m$

```
Require: $\Upsilon \subseteq \hat{X}_1, l, \mathbf{C}$
 1: if $l = L$ then
 2:     $\langle W, C\rangle \leftarrow$ REACH$_\infty(\Gamma_{L,1}(\Upsilon), l)$
 3:     $\mathbf{C} \leftarrow \mathbf{C} \cup \{C\}$
 4:     $\Upsilon \leftarrow \Upsilon \cup \Gamma_{1l}(W)$ // Save $W$ to $\Upsilon$
 5:     if $L = 1$ then    // Single-layered reachability
 6:         return $\langle \Upsilon, \mathbf{C}\rangle$
 7:     else
 8:         $\langle \Upsilon, \mathbf{C}\rangle \leftarrow$ REACHIT$_m(\Upsilon, l-1, \mathbf{C})$
 9:         return $\langle \Upsilon, \mathbf{C}\rangle$
10:     end if
11: else
12:     $\langle W, C\rangle \leftarrow$ REACH$_m(\Gamma_{l,1}(\Upsilon), l)$
13:     $\mathbf{C} \leftarrow \mathbf{C} \cup \{C\}$
14:     $\Upsilon \leftarrow \Upsilon \cup \Gamma_{1l}(W)$ // Save $W$ to $\Upsilon$
15:     if Fixed-point is reached in line 12 then
16:         if $l = 1$ then // Finest layer reached
17:             return $\langle \Upsilon, \mathbf{C}\rangle$
18:         else    // Go finer
19:             $\langle \Upsilon, \mathbf{C}\rangle \leftarrow$ REACHIT$_m(\Upsilon, l-1, \mathbf{C})$
20:             return $\langle \Upsilon, \mathbf{C}\rangle$
21:         end if
22:     else    // Go coarser
23:         $\langle \Upsilon, \mathbf{C}\rangle \leftarrow$ REACHIT$_m(\Upsilon, l+1, \mathbf{C})$
24:         return $\langle \Upsilon, \mathbf{C}\rangle$
25:     end if
26: end if
```

Sec. 3.3 is equivalent to a quantizer defined via the ranking induced by the induction depth $d$.

To see why (26) holds, recall that, for any $x \in \hat{Q}_1^{-1}(\mathrm{dom}(C))$, every trajectory $\xi \in \mathscr{B}(\vec{S}_1^{cl})|_X$ with $\xi(0) = x$ will reach (the projection of) the target $T$, and that Alg. 1 will eventually reach $l = 1$. Now assume that Alg. 1 was run until depth $d$ where $l = 1$ and let $k' \in \mathrm{dom}(\xi)$ be s.t. for all $k > k'$ with $k \in \mathrm{dom}(\xi)$, $\xi(k) \in \bigcup_{d'<d} \mathbf{Q}^{-1}(\mathrm{dom}(C^{d'}))$ while this is not true for $\xi(k')$. Given that $l_d = 1$, we execute REACH for $l = 1$ implying that $\xi(k')$ up to $\xi(k' - m)$ will be added to the domain of controller $C^d$ and saved in $\Upsilon$. As $x \in \hat{Q}_1^{-1}(\mathrm{dom}(C))$, Alg. 1 can only terminate if $\hat{Q}_1(\xi(0))$ is eventually reached. Therefore, we can apply the above argument iteratively to prove the statement.

## 4.2. Safety

Note that the ABCD method already incorporates the possibility to restrict any control problem to a designated safe region $X'$ of the state space $X \supset X'$ by constructing the abstract system such that it blocks for every abstract state in $\hat{X} \setminus \hat{X}'$. For the problem of reaching a target region $R \subseteq \hat{X}'$ while staying safely inside $\hat{X}'$, this simplifies to removing all abstract states $\hat{X} \setminus \hat{X}'$ before running the algorithm for reachability. This feature was implicitly used in the example of Sec. 1. However, safety control problems may arise as sub-problems in other synthesis tasks, and so we now give an algorithm for multi-layered synthesis for safety properties.

**Algorithm 2** Procedure SAFEIT

```
Require: $\Upsilon \subseteq \hat{X}_1$, $\Upsilon' \subseteq \hat{X}_1$, $l$, $\mathbf{C}$
 1: $W \leftarrow \mathrm{Pre}_{\hat{S}_l}(\Gamma_{l1}(\Upsilon)) \cap \Gamma_{l1}(\Upsilon)$
 2: $\mathbf{C} \leftarrow \mathbf{C} \cup \{C_l \leftarrow (W, \hat{U}, \emptyset)\}$
 3: $\Upsilon' \leftarrow \Upsilon' \cup \Gamma_{1l}(W)$
 4: if $l \neq 1$ then
 5:     SAFEIT($\Upsilon, \Upsilon', l-1, \mathbf{C}$) // go finer
 6: else
 7:     if $\Upsilon \neq \Upsilon'$ then
 8:         SAFEIT($\Upsilon', \emptyset, L, \emptyset$) // start new iteration
 9:     else
10:         return $\langle \Upsilon, \mathbf{C} \rangle$ // terminate
11:     end if
12: end if
```

It is common practice in ABCD to ensure safety for sampling times only. This implicitly assumes that sampling times and grid sizes are chosen such that no "holes" occur between consecutive cells visited in a trajectory. This can be formalized by additional assumptions on the growth rate of the flow.

**Single-Layered Control** We consider a safety control problem $\langle \hat{\mathbf{S}}, \Box T \rangle$ and recall how it is commonly solved by ABCD for $L = l = 1$. In this case one iteratively computes the sets

$$W^0 = \widehat{T} \text{ and } W^{i+1} = \mathrm{Pre}_{\hat{S}_1}(W^i) \cap \widehat{T} \tag{27}$$

until an iteration $N \in \mathbb{N}$ with $W^N = W^{N+1}$ is reached, where $\widehat{T}$ is given by (21). Then $C$ with $\mathrm{dom}(C) = W^N$, and

$$\hat{u} \in C(\hat{x}) \Rightarrow \hat{F}_1(\hat{x}, \hat{u}) \subseteq \mathrm{dom}(C) \tag{28}$$

for all $\hat{x} \in \mathrm{dom}(C)$, is known to be a safety controller for $\langle \hat{\mathbf{S}}, \Box T \rangle$.

Thus, the above synthesis algorithm is sound. However, completeness is not guaranteed; there may exist a state $x \in X$ s.t. $\hat{Q}_1(x) \notin \mathrm{dom}(C)$, and there may exist a controller $C'$ based on a finer abstraction, solving the safety control problem $\langle \hat{\mathbf{S}}, \Box T \rangle$ s.t. $\hat{Q}_1(x) \in \mathrm{dom}(C')$.

**Multi-Layered Systems.** Given a sequence of $L$ abstract systems $\hat{\mathbf{S}} := \{\hat{S}_l\}_{l \in [1;L]}$ we now present a multi-layered safety algorithm formalized by the iterative function SAFEIT given as pseudo-code in Alg. 2.

When initialized with SAFEIT($\widehat{T}_1, \emptyset, L, \emptyset$), Alg. 2 performs the following computations: it starts in layer $l = L$ with an outer recursion count $i = 1$ (not shown in Alg. 2) and reduces $l$, one step at the time, until $l = 1$ is reached, at which point it then starts over again from layer $L$ with $i = i + 1$ and a new safe set $\Upsilon'$. In every such iteration $i$, one step of the safety fixed-point is performed for every layer and the resulting set is stored in the layer 1 map $\Upsilon' \subseteq \hat{X}_1$, whereas $\Upsilon \subseteq \hat{X}_1$ keeps the knowledge of the previous iteration. If the finest layer is reached and we have $\Upsilon = \Upsilon'$, the algorithm terminates. Otherwise $\Upsilon'$ is copied to $\Upsilon$, $\Upsilon'$ and $\mathbf{C}$ are reset to $\emptyset$ and SAFEIT starts a new iteration (see line 8).

After SAFEIT has terminated, it returns a multi-layered controller $\mathbf{C} = \{C^l\}_{l\in[1;L]}$ whose domains are given by the winning regions (see line 10), and the control actions can be computed by choosing one input $\hat{u} \in \hat{U}$ for every $\hat{x} \in \mathrm{dom}(C^l)$ s.t.

$$\hat{u} = C^l(\hat{x}) \Rightarrow \hat{F}_l(\hat{x},\hat{u}) \subseteq \Gamma_{l1}(\Upsilon). \tag{29}$$

Note that states encountered for layer $l$ in iteration $i$ are saved to the lowest layer 1 (line 3 of Alg. 2) and "loaded" back to the respective layer $l$ in iteration $i+1$ (line 1 of Alg. 2). Therefore, a state $\hat{x} \in \hat{X}_l$ with $l > 1$, which was not contained in $W$ as computed in layer $l$ and iteration $i$ via line 1 of Alg. 2, might still be included in $\Gamma_{l1}(\Upsilon)$ loaded in the next iteration $i+1$ when re-computing line 1 for $l$. This happens if all states $x \in \hat{x}$ were added to $\Upsilon'$ by some layer $l' < l$ in iteration $i$. This allows the algorithm to "bridge" regions that require a finer grid and to use layer $L$ in all remaining regions of the state space.

**Soundness and Relative Completeness** Due to the effect described above, the map $W$ encountered in line 1 for a particular layer $l$ throughout different iterations $i$ might not be monotonically shrinking. However, the latter is true for layer 1. Indeed, one can show [HMMS18b] that $\mathbf{C}$ is sound and relatively complete w.r.t. single-layer control for layer $l = 1$.

## 5. Further Reading

A good, but perhaps already dated, introduction to abstraction-based control is Tabuada's book [Tab09]. Our exposition follows our own recent work [HMMS18c,HMMS18b].

The original techniques for abstraction-based control relied on $\varepsilon$-alternating bisimulation relations [GP07,Gir12,PGT08,GPT10]. These relations, when they exist, allow proving an "if and only if" result: a controller can be synthesized in the abstraction iff one exists in the original system. The notion of *feedback refinement relations* (FRR) was introduced in [RWR17], and strengthened the notion of alternating simualtion relations [AHKV98] to the setting of continuous control. The same paper shows how to compute growth bounds for a non-linear system—this is the basis for the SCOTS [RZ16] and MASCOT tools. Similar ideas are implemented in CoSyMa [MGG13], ROCS [LL18], and abstr-refinement [BNO18]. More recently, the PFACES tool [KZ19] scales up ABCD through parallelization and GPU-based techniques.

The automata-theoretic underpinnings of ABCD use algorithms for reactive synthesis for $\omega$-regular specifications [EJ91,Tho95,MPS95].

ABCD continues to be an active research direction, with new results targeting the expressivity and scalability of the method. Over the years, the curse of dimensionality has proven to be the main computational bottleneck for application of ABCD in real-world problems. As the number of system variables increases, the discretization step produces exponentially many states or inputs. There have been many heuristic approaches in the past which have dealt with this scalability issue for ABCD. Broadly, they can be classified into two categories: the first uses the system specification to limit the computational effort, and the second uses the system dynamic model to do the same. Our multi-layered ABCD technique falls in the first category.

Multi-layered algorithms were proposed by Girard et al. [CGG11a,CGG11b] (recently revisited in [GGM16]) in the special case of safety and reachability control of *unperturbed* switched systems (our treatment, in contrast, has a disturbance controlled by an adversary).

Subsequently, multi-layered ABCD was implemented in a *lazy* way, by selectively computing the abstraction of finer layers of abstraction as needed, rather than up front [HMMS18b,HMMS18a]. The idea there is that we start with a fully computed coarsest layer of abstraction, and locally switch to the finer layers only when the synthesis cannot progress anymore in the coarser layer. Similar techniques also appeared independently in the context of multi-resolution abstractions [NO14,BNO18]. An orthogonal approach is to lazily compute the abstraction by only partially and incrementally reavealing the inputs while computing the transitions [HT18].

The second approach for improving scalability of ABCD is to exploit the structure of the underlying system model. One notable technique in this category uses the decomposed structure of a given system to compute local abstractions and controllers [TI08,MSSM17,MSSM18]. While the decomposition-based technique addresses systems with many loosely coupled components, there are recent approaches which can even use the loose coupling between different variables of a monolithic system component [GKA17]. Another example of such a technique is the fast but relatively imprecise abstraction technique for monotone systems [KAS17].

**Acknowledgements.** We thank our collaborators on the `MASCOT` project: Yunjun Bai, Kyle Hsu, Sadegh Soudjani, and Damien Zufferey.

## References

[AHKV98] R. Alur, T.A. Henzinger, O. Kupferman, and M.Y. Vardi. Alternating refinement relations. In *CONCUR'97*, LNCS 1466, pages 163–178. Springer, 1998.

[BNO18] Oscar Lindvall Bulancea, Petter Nilsson, and Necmiye Ozay. Nonuniform abstractions, refinement and controller synthesis with novel bdd encodings. *IFAC-PapersOnLine*, 51(16):19–24, 2018.

[BS06] J. Bradfield and C. Stirling. Modal mu-calculi. In *The Handbook of Modal Logic*, pages 721––756. Elsevier, 2006.

[CGG11a] J. Cámara, A. Girard, and G. Gössler. Safety controller synthesis for switched systems using multi-scale symbolic models. In *CDC '11*, pages 520–525, 2011.

[CGG11b] J. Cámara, A. Girard, and G. Gössler. Synthesis of switching controllers using approximately bisimilar multiscale abstractions. In *HSCC*, pages 191–200, 2011.

[EJ91] E. A. Emerson and C. S. Jutla. Tree automata, mu-calculus and determinacy. In *FOCS'91*, pages 368–377, 1991.

[GGM16] A. Girard, G. Gössler, and S. Mouelhi. Safety controller synthesis for incrementally stable switched systems using multiscale symbolic models. *TAC*, 61(6):1537–1549, 2016.

[Gir12] Antoine Girard. Controller synthesis for safety and reachability via approximate bisimulation. *Automatica*, 48(5):947–953, 2012.

[GKA17] Felix Gruber, Eric S Kim, and Murat Arcak. Sparsity-aware finite abstraction. In *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, pages 2366–2371. IEEE, 2017.

[GP07] A. Girard and G. J. Pappas. Approximation metrics for discrete and continuous systems. *TAC*, 25(5):782–798, 2007.

[GPT10] A. Girard, G. Pola, and P. Tabuada. Approximately bisimilar symbolic models for incrementally stable switched systems. *TAC*, 55(1):116–126, 2010.

[HMMS18a] Kyle Hsu, Rupak Majumdar, Kaushik Mallik, and Anne-Kathrin Schmuck. Lazy abstraction-based control for reachability. *arXiv preprint arXiv:1804.02722*, 2018.

[HMMS18b] Kyle Hsu, Rupak Majumdar, Kaushik Mallik, and Anne-Kathrin Schmuck. Lazy abstraction-based control for safety specifications. In *2018 IEEE Conference on Decision and Control (CDC)*, pages 4902–4907. IEEE, 2018.

[HMMS18c] Kyle Hsu, Rupak Majumdar, Kaushik Mallik, and Anne-Kathrin Schmuck. Multi-layered abstraction-based controller synthesis for continuous-time systems. In *HSCC*, pages 120–129. ACM, 2018.

[HT18] Omar Hussien and Paulo Tabuada. Lazy controller synthesis using three-valued abstractions for safety and reachability specifications. In *2018 IEEE Conference on Decision and Control (CDC)*, pages 3567–3572. IEEE, 2018.

[KAS17] Eric S Kim, Murat Arcak, and Sanjit A Seshia. Symbolic control design for monotone systems with directed specifications. *Automatica*, 83:10–19, 2017.

[KZ19] Mahmoud Khaled and Majid Zamani. pFaces: an acceleration ecosystem for symbolic control. In *HSCC 2019*, pages 252–257. ACM, 2019.

[LL18] Yinan Li and Jun Liu. Rocs: A robustly complete control synthesis tool for nonlinear dynamical systems. In *Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (part of CPS Week)*, pages 130–135. ACM, 2018.

[MGG13] Sebti Mouelhi, Antoine Girard, and Gregor Gössler. Cosyma: a tool for controller synthesis using multi-scale abstractions. In *HSCC*, pages 83–88. ACM, 2013.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer, 1992.

[MPS95] O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In *STACS'95*, volume 900 of *LNCS*, pages 229–242. Springer, 1995.

[MSSM17] Kaushik Mallik, Sadegh Esmaeil Zadeh Soudjani, Anne-Kathrin Schmuck, and Rupak Majumdar. Compositional construction of finite state abstractions for stochastic control systems. In *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, pages 550–557. IEEE, 2017.

[MSSM18] Kaushik Mallik, Anne-Kathrin Schmuck, Sadegh Soudjani, and Rupak Majumdar. Compositional synthesis of finite state abstractions. *IEEE Transactions on Automatic Control*, 2018.

[NO14] Petter Nilsson and Necmiye Ozay. Incremental synthesis of switching protocols via abstraction refinement. In *53rd IEEE Conference on Decision and Control*, pages 6246–6253. IEEE, 2014.

[PGT08] G. Pola, A. Girard, and P. Tabuada. Approximately bisimilar symbolic models for nonlinear control systems. *Automatica*, 44(10):2508–2516, 2008.

[RWR17] G. Reissig, A. Weber, and M. Rungger. Feedback refinement relations for the synthesis of symbolic controllers. *TAC*, 62(4):1781–1796, 2017.

[RZ16] M. Rungger and M. Zamani. SCOTS: A tool for the synthesis of symbolic controllers. In *HSCC'16*, pages 99–104. ACM, 2016.

[Tab09] P. Tabuada. *Verification and control of hybrid systems: a symbolic approach*. Springer, 2009.

[Tho95] Wolfgang Thomas. On the synthesis of strategies in infinite games. In *STACS'95 Munich, Germany*, pages 1–13. 1995.

[TI08] Yuichi Tazaki and Jun-Ichi Imura. Bisimilar finite abstractions of interconnected systems. *HSCC'08*, pages 514–527, 2008.

[WRR16] A. Weber, M. Rungger, and G. Reissig. Optimized state space grids for abstractions. *TAC*, 2016.

This page intentionally left blank

*Engineering Secure and Dependable Software Systems*
*A. Pretschner et al. (Eds.)*
*IOS Press, 2019*
*© 2019 The authors and IOS Press. All rights reserved.*
*doi:10.3233/978-1-61499-977-5-137*

# The Thousand-and-One Cryptographers

Annabelle MCIVER [a,1], and Carroll MORGAN [b]
[a] *Dept. of Computing, Macquarie University, NSW 2109 Australia*
[b] *School of Comp. Sci. and Eng., Univ. NSW, NSW 2052 Australia and Data61*

**Abstract.** Chaum's *Dining Cryptographers* protocol crystallises the essentials of security just as other famous diners once captured deadlock and livelock: it is a benchmark for security models and their associated verification methods.

Here we give a correctness proof of the Cryptographers in a new style, one in which stepwise refinement plays a prominent role. Furthermore, our proof applies to arbitrarily many Diners: to our knowledge we are only the second group to have done that.

The proof is based on the *Shadow Security Model* which integrates non-interference and program refinement: with it, we try to make a case that stepwise development of security protocols is not only possible but actually is quite a good idea. It benefits from more than three decades' of experience of how layers of abstraction can both simplify the design process and make its outcomes more likely to be correct.

**Keywords.** Security, refinement, automated proof, correctness, unbounded dining cryptographers.

## 1. Introduction: refinement of security properties

Program development by stepwise refinement [34] is widely accepted as a good idea in theory, but it is often a late arrival in practice. Indeed, with some notable exceptions [1,5] most current approaches and tools for correctness concentrate on proving [2] that a single system has certain desirable properties, whereas a refinement-based approach would rather prove that one (real, i.e. implementation) system had all the desirable properties of another (ideal, i.e. specification) system.

As an example, we note the frequent claims that downgrading is a challenging issue in the non-interference model of security [12]. For example, in that model a program is secure if observation of its "low-security" visible outputs does not reveal anything about its "high-security" hidden inputs; thus in the context of visible integer variables $v$ and hidden $h$ the program $v{:=}\,0{\times}h$ is secure but $v{:=}\,1{\times}h$ is not. As an intermediate option there is the program $v{:=}\,h{\div}2$ that "downgrades" the security, revealing in this case most bits of $v$ but not all, yet it is not considered to be "intermediately" secure: like $v{:=}\,1{\times}h$, it is considered (simply) insecure.

[1]Corresponding Author: Dept. of Computing, Macquarie University, NSW 2109 Australia; E-mail: annabelle.mciver@mq.edu.au.
[2]We include model checking as a form of proving, at this informal level.

Now in a real system we might find the code

$$v{:=}\,0;\;\; \textbf{while}\; v{+}2 \le h\; \textbf{do}\; \langle \textit{send two bytes} \rangle ;\, v{:=}\, v{+}2\; \textbf{end}\;, \qquad (1)$$

in which $v$ counts the number of bytes sent, ensuring no more than $h$ can be sent overall. If the sent messages are observed and counted, then this program also reveals –to anyone aware of the source code– all but the low-order bit of $h$. Like $v{:=}\, h \div 2$ above, it is considered insecure in the non-interference model.

Downgrading is inescapable in practice, and it is to reason about it effectively –in spite of the black-or-white judgement of the original non-interference model– that downgrading extensions to that model are introduced [7,20] in which one can express, by annotations of the code, the information leaks that are to be considered acceptable. The proof of correctness is then relative to those annotations.

But there is an alternative to concentrating on downgrading exclusively: instead we concentrate on refinement, with downgrading then a special case. With this approach we would describe a downgrading policy for the loop of (1) as a refinement, saying that

$$v{:=}\, 2(h{\div}2) \quad \sqsubseteq \quad v{:=}\,0;\;\; \textbf{while}\; v{+}2 \le h\; \textbf{do}\; \langle \textit{send} \rangle ;\, v{:=}\, v{+}2\; \textbf{end}\;, \qquad (2)$$

i.e. that the *lhs* "is refined by" the *rhs* and meaning that the desirable properties of the specification on the left –including its not revealing $h$'s low-order bit– are shared by its implementation on the right. The downgrading aspect is that the *rhs* can indeed reveal the higher-order bits of $h$ — because the *lhs* does just that. We feel that a refinement-based approach has many advantages, demonstrated over the years by its success generally (where it has after all been adopted). In this particular case, for example, using it would mean that we require neither annotations nor an explicit notion of downgrading.

Integrating refinement and security is exactly what we do here: we make (2) precise by giving an appropriately extended definition of refinement, one which is "security aware." It is explained below (and earlier [25,27]). In doing so we join a small number of other researchers –from the large security community– who have similar aims [21,2,9]; we compare our work with theirs in Sec. 9. Some conspicuous aspects of our approach are as follows.

Refinement is complementary to abstraction, and abstraction can be viewed in turn as demonic nondeterministic choice resolved at design-time; but it is well known that there are conceptual benefits to conflating abstraction with run-time demonic choice [8,4,17,23] and so it is natural to include demonic choice in our security model in both the design- and run-time senses. Where this has been done by others, in some cases the non-interference model has been extended so that the "full range of nondeterminism" of the hidden variables must not be dependent on visible variables' observed values, but the nondeterminism cannot subsequently be reduced as refinement would ordinarily allow [19,32]; and in other cases the requirement has been imposed that –while nondeterminism is allowed in the model– in the final implementation program there must be none of it remaining [31]. *One* *aspect* of our work is that, in contrast, we include both features: nondeterminism can be reduced; and some of it can remain in the implementation. This requires careful treatment of hidden- vs. visible nondeterminism, and is how we solved the Refinement Paradox [24,18].

*A second aspect* is that our notion of adversary is quite strong: we allow perfect recall [11], that intermediate values of visible variables can be observed even if they subsequently are overwritten; and we allow an attacker's observation of control flow, that conditionals' Booleans expressions are (implicitly) leaked. We assume also that the program code is known. Doing all this effectively, while avoiding an infinite regress to attacks at the level of quarks, requires explicit treatment of atomicity at some point. We define that.

The reason for the strong adversary is that refinements must be effective locally: refinement of a small fragment in a large program must refine the whole program even if the refinement was proved only for the fragment. This is monotonicity — and without it no scaled-up development is possible. Since for some fresh local visible variable $v'$ it is a refinement to insert assignment $v'{:=}\,v$ willy-nilly at almost any place in a program, we must live with the fact that $v$'s value at that point will possibly be preserved (in some local $v'$) in spite of $v$'s subsequent overwriting: that is, although the unfortunate $v'{:=}\,v$ might not be there "now," one developer must accept that a second developer in some other building might put it there "later" without asking. After all, if it's a refinement (and it is) then he doesn't *need* to ask.

Rather than make refinements unworkable, rather the strong-adversary assumptions make them more applicable. Distributed protocols (such as the Dining Cryptographers) can be treated as single "sequential" programs because the information hiding normally implied by non-interference's end-to-end analysis does not apply. Indeed, if it did, the sequential formulation would seem to be hiding the transfer of messages and the interleaving of concurrent threads, and that would make it unsound. For example, if Agent $A$ executes $v{:=}\,h$ and Agent $B$ then executes $v{:=}\,0$ we can analyse this as the single sequential program $v{:=}\,h; v{:=}\,0$ without having accidentally (and incorrectly) ignored the fact that $A$ can observe $v$ (and hence learn $h$) before $B$'s thread has begun the execution that would overwrite it.

*A third aspect* of our approach is that we concentrate on algebraic reasoning for proving refinement: although we do have both an operational model (Sec. 2) and a language of logical assertions (Sec. 3) for refinement-based security, we use those mainly for proving schematic program-fragment -equalities and refinements (Sec. 4). Those schemes, rather than the logic or the model, are then what is used in the derivation of specific programs. For this we need a program-level indication of information escape, analogous to the way in which the *assert* statement can embed Hoare-Logic pre- and post-conditions within program code [16,23,35]. This is our **reveal** $E$ statement that publishes $E$'s value for all to read, but does not change any program variable: its purpose is to bring an extra expressivity that helps formulate general algebraic (in)equalities. Since functionally it acts as **skip** on program variables (having no effect at all), but *wrt* secrecy it does not act as **skip** (it releases information but **skip** does not), its behaviour considered alone will capture much of the flavour of what we intend to do.

Section 2 gives our relational-style operational model; Section 3 describes a corresponding modal logic based on the logic of knowledge; and Section 4 introduces our program algebra. Sections 5–8 demonstrate the approach on examples of increasing complexity.

Throughout we use left-associating dot for function application, so that $f.x.y$ means $(f(x))(y)$ or $f(x,y)$, and comprehensions/quantifications are written uniformly, as$(\mathsf{Q}x{:}\,T \mid R \cdot E)$ for quantifier $\mathsf{Q}$, bound variable(s) $x$ of type(s) $T$, range predicate $R$ (probably) constraining $x$ and element-constructor $E$ in which $x$ (probably) appears free. For sets the opening "($\mathsf{Q}$" is "{" and the closing ")" is "}" so that e.g. the comprehension $\{x,y{:}\,\mathbb{N} \mid y = x^2 \cdot z{+}y\}$ is the set of all natural numbers that exceed $z$ by a perfect square exactly, that is $\{z, z{+}1, z{+}4, \cdots\}$.

## 2. The Shadow model of security

### 2.1. Introduction; non-interference; logic of knowledge

Our operational model is loosely based on non-interference [12] and on the Kripke structures associated with the (modal) Logic of Knowledge [11]: it extends the former with concepts of the latter, and is targetted specifically at development of secure programs (in its terms) via a process of stepwise refinement. The "shadow" of the title refers to an extra semantic component that tracks a postulated attacker's inferred knowledge, or ignorance, of hidden (high-security) variables.

The non-interference approach (in its simplest form) partitions variables into high-security- and low-security classes: we call them *hidden* and *visible* respectively. A "non-interference -secure" program then prevents an attacker's inferring hidden variables' initial values from initial and/or final visible variables' values. Assuming for simplicity just two variables $v, h$ of class visible, hidden resp. we consider in this simple approach a possibly nondeterministic program $r$ that takes initial states $(v, h)$ to sets of final visible states $v'$ and is thus of type $\mathcal{V} \to \mathcal{H} \to \mathbb{P}\mathcal{V}$, where $\mathcal{V}, \mathcal{H}$ are the value sets corresponding to the types of $v, h$ respectively; note that we are ignoring final hidden values at this point. Such a program $r$ is *non-interference -secure* just when for any initial visible value the set of possible final visible values is independent of the initial hidden value [19,32,28]:

$$\big(\forall v{:}\,\mathcal{V}; h_0, h_1{:}\,\mathcal{H} \cdot r.v.h_0 = r.v.h_1\big) \ .$$

Our first extension of the simple approach is to concentrate on final- (rather than initial) hidden values and therefore model programs by the slightly more elaborate type $\mathcal{V} \to \mathcal{H} \to \mathbb{P}(\mathcal{V} \times \mathcal{H})$. For two such programs $r_{\{1,2\}}$ we say that $r_1 \sqsubseteq r_2$, that $r_1$ "is (securely) refined by" $r_2$, just when the following both hold:

(i) For any initial state $v, h$ each possible $r_2$ outcome is also a possible $r_1$ outcome, that is

$$\big(\forall v{:}\,\mathcal{V}; h{:}\,\mathcal{H} \cdot r_1.v.h \supseteq r_2.v.h\big) \ .^3$$

This is the normal "can reduce nondeterminism" form of refinement, i.e. it is the classical form.

(ii) For any initial state $v, h$ and final state $v'$ possible for $r_2$ (i.e. for which there exists a compatible $h'_2$), each $h'_1$ that $r_1$ can produce with that $v, v'$ can also be produced by $r_2$ with that same $v, v'$, that is

$$\left(\begin{array}{ll}\forall v, v' \colon \mathcal{V}; h, h'_1, h'_2 \colon \mathcal{H} \cdot & (v', h'_2) \in r_2.v.h \wedge (v', h'_1) \in r_1.v.h \\ & \Rightarrow (v', h'_1) \in r_2.v.h\end{array}\right) .$$

This second condition says that for any particular visible final $v'$ the attacker's "ignorance" of $h'$'s compatible with that $v'$ cannot be decreased by the refinement from $r_1$ to $r_2$: whenever we must consider out of ignorance that some $h'_1$ is possible for $r_1$, we must be forced to consider that same $h'_1$ to be possible for $r_2$ as well. This extended "secure" refinement is thus more restrictive than the classical.

In fact, in this moderately extended approach, the two conditions (i), (ii) together do not allow ignorance strictly to increase: refinement then boils down to a simple policy of allowing decrease of nondeterminism in $v$ but not in $h$. But strict increase of hidden nondeterminism will be possible (3) in the more ambitious approach we introduce below.

As an example of the above we restrict all our variables' types so that $\mathcal{V} = \mathcal{H} = \{0, 1\}$, and we let $r_1$ be the maximally nondeterministic program that can produce from any initial values $(v, h)$ any one of the four possible $(v', h')$ final values in $\mathcal{V} \times \mathcal{H}$. Then the program $r_2$ that produces only the two final values in $\{(0, 0), (0, 1)\}$ is a refinement of $r_1$ that strictly reduces nondeterminism in $v$ but not in $h$, and is (therefore) still secure. But the program $r'_2$ that produces only the two final values in $\{(0, 0), (1, 1)\}$ is not a secure refinement, because it reduces nondeterminism in $h$ (as well).

Thus $r_1$ allows any behaviour, and $r_2$ simply reduces the nondeterminism by limiting its outputs to $v'{=}0$ only; but, even with the limited outputs, an attacker of $r_2$ can gain no more knowledge of $h'$'s value than it would have had from attacking $r_1$ instead. So $r_1 \sqsubseteq r_2$. An attacker of $r'_2$ however can deduce $h'$'s value from having seen $v'$'s, since the program guarantees they will be equal. Since that attack is not possible on $r_1$, we have $r_1 \not\sqsubseteq r'_2$.

## *2.2. The Shadow of h*

In $r_1$ above, when the final value $v'$ was 0 the corresponding set of associated possible values of $h'$ was $\{0, 1\}$. This set $\{0, 1\}$ is called "The Shadow," and represents explicitly an attacker's ignorance of the hidden variables' values. In $r_2$ that shadow was the same (for $v' = 0$); but in $r'_2$ the shadow was smaller, and that is why we don't consider $r'_2$ to be a refinement of $r_1$ as far as security is concerned.

[3]Some researchers [2] do not consider the final $h$ here: for our purposes that would make our program operators non-monotonic for refinement (thus a failure of compositionality). That is, if for hidden $h$ the assignments $h{:=}\,0$ and $h{:=}\,1$ are the same, then for visible $v$ the compositions $h{:=}\,0; v{:=}\,h$ and $h{:=}\,1; v{:=}\,h$ should not be different.

In the shadow semantics we model this explicit ignorance-set, so that our final program state is extended to a triple $(v', h', H')$ with $H'$ a subset of $\mathcal{H}$ — in each triple the $H'$ contains exactly those (other) values that $h'$ might have had. The (extended) output-triples of the three example programs are then respectively

$$\begin{array}{ll} r_1 — & \{(0,0,\{0,1\}),(0,1,\{0,1\}),(1,0,\{0,1\}),(1,1,\{0,1\})\} \\ r_2 — & \{(0,0,\{0,1\}),(0,1,\{0,1\})\} \\ r_2' — & \{(0,0,\{0\}), \quad (1,1,\{1\})\}\ , \end{array}$$

and we can see $r_1 \sqsubseteq r_2$ because $r_1$'s set of outcomes includes all of $r_2$'s. But e.g. the outcome $(0,0,\{0\})$ of $r_2'$ does not occur among $r_1$'s outcomes, nor is there even an $r_1$-outcome $(0,0,H')$ with $H' \subseteq \{0\}$ that would satisfy (ii). That is why we say that $r_1 \not\sqsubseteq r_2'$ for security.

Now –the final enhancement– for sequential composition of shadow-enhanced programs also initial triples $(v,h,H)$ must be dealt with, since the final triples of a first component become initial triples for the second. We therefore define the full shadow semantics, in the next subsection, by showing how those triples are related by program execution.

## *2.3. The Shadow semantics of atomic programs*

A "non-shadow," call it *classical* program $r$ is effectively an input-output relation between $\mathcal{V} \times \mathcal{H}$ -pairs. Its shadow version $\langle r \rangle$ is a relation between $\mathcal{V} \times \mathcal{H} \times \mathbb{P}\mathcal{H}$ -triples and is defined as follows:

**Definition 1.** *Atomicity Given a standard program* $r{:}\, \mathcal{V} \to \mathcal{H} \to \mathbb{P}(\mathcal{V} \times \mathcal{H})$ *we define its* atomic shadow version $\langle r \rangle{:}\, \mathcal{V} \to \mathcal{H} \to \mathbb{P}\mathcal{H} \to \mathbb{P}(\mathcal{V} \times \mathcal{H} \times \mathbb{P}\mathcal{H})$ *so that* $\langle r \rangle.v.h.H \ni (v', h', H')$ *just when*

*(i) we have both* $r.v.h \ni (v', h')$
*(ii) and* $H' = \{h'{:}\,\mathcal{H} \mid (\exists h{:}\, H \cdot r.v.h \ni (v', h'))\}$ .

*The final shadow component is generated from the initial shadow component and any nondeterminism present in the program.*

As a first example, let the syntax $x{:\in}\, S$ denote the standard program that chooses variable $x$'s value from a set $S$, which we assume to be non-empty. From Def. 1 we have that

(i) A choice of visible $v$ has no effect on $h, H$
because $\langle v{:\in}\, S \rangle.v.h.H = \{v'{:}\, S \cdot (v', h, H)\}$ , but
(ii) A choice of hidden $h$ introduces ignorance
because $\langle h{:\in}\, S \rangle.v.h.H = \{h'{:}\, S \cdot (v, h', S)\}$ and finally
(iii) An assignment of hidden to visible "collapses" any ignorance that might be there because $\langle v{:=}\, h \rangle.v.h.H = \{(h, h, \{h\}\}$ .

From (ii) and (iii) above we can therefore see that in the sequential composition $\langle h{:\in}\, S \rangle; \langle v{:=}\, h \rangle$ the first statement introduces ignorance –we do not know $h$'s exact value "at the semicolon"– but the second statement then removes it because we can deduce $h$'s value, at the end, by observing $v$. The composition as a whole is nondeterministic, because $v, h$'s common final value is drawn arbitrarily from $S$; but the nondeterminism is observable.

In general atomicity is not preserved by composition (indeed one expects it not to be); but in many simple cases it is preserved.

**Lemma 1.** *atomicity and composition* Given two programs $r_{\{1,2\}}$ over $v, h$ we have $\langle r_1; r_2\rangle = \langle r_1\rangle; \langle r_2\rangle$ just when $v$'s *intermediate* value, i.e. "at the semicolon," can be deduced from its *endpoint* values, i.e. initial and final, possibly in combination. The semicolon denotes relational composition in both cases, of pairs on the left and of triples on the right.

*Proof:* Given in [22]. □

In fact this lemma is more significant when its conditions are *not* met than when they are. It means for example that we cannot conclude from Lem. 1 that $\langle v{:=}\,h; v{:=}\,0\rangle = \langle v{:=}\,h\rangle; \langle v{:=}\,0\rangle$, since on the left the intermediate value of $v$ cannot be deduced from its endpoint values: for $h$ is not visible at the beginning and $v$ itself has been "erased" at the end. And indeed from Def. 1

(i) On the left we have $\langle v{:=}\,h; v{:=}\,0\rangle.v.h.H = \{(0, h, H)\}$, whereas
(ii) On the right we have $(\langle v{:=}\,h\rangle; \langle v{:=}\,0\rangle).v.h.H = \{(0, h, \{h\})\}$ .

This phenomenon is called *perfect recall* [11] –that $v$'s temporary receipt of $h$ is seen by an attacker even though it is subsequently overwritten– and it is a feature (not a bug). It is due to our refinement-oriented point of view, as we now explain.

## *2.4. Refinement vs. atomicity:* Gedanken *experiments*

Perfect recall is necessary because refinement must be *monotonic*, i.e. (A) that refinement of a program portion must refine the whole program; and we insist additionally (B) that classical refinements involving $v$ only must remain valid. Both principles (A,B) are required in order to be able to develop large programs via local reasoning over small portions.

For example, without perfect recall the overwriting of $v$ would prevent program $v{:=}\,h; v{:\in}\,\{0,1\}$ from revealing $h$. Yet from (B) we have $v{:\in}\,\{0,1\} \sqsubseteq v{:=}\,v$; and then from (A) we have $(v{:=}\,h; v{:\in}\,\{0,1\}) \sqsubseteq (v{:=}\,h; v{:=}\,v)$ — and it is a contradiction of secure refinement that the *lhs* does not reveal $h$ but the *rhs* does. Thus the premise –that recall is not perfect– is false.

There is a similar experiment for conditionals: because (A,B) imply the refinement

$$\begin{array}{ll} & \textbf{if } h{=}0 \textbf{ then } v{:\in}\,\{0,1\} \textbf{ else } v{:\in}\,\{0,1\} \textbf{ fi} \\ \sqsubseteq & \textbf{if } h{=}0 \textbf{ then } v{:=}\,0 \qquad\;\; \textbf{else } v{:=}\,1 \qquad\;\; \textbf{fi} \ , \end{array}$$

we must accept that the **if**-test reveals its outcome, in this case whether $h{=}0$ holds. And nondeterministic choice $P_1 \sqcap P_2$ is visible to the attacker because each of the two branches $P_{\{1,2\}}$ can be refined separately in a similar way.

| | Program $P$ | Semantics $[\![P]\!].v.h.H$ | |
|---|---|---|---|
| Publish a value | $\textbf{reveal}\ E.v.h$ | $\{\ (v,\ h,\ \{h'{:}\,H \mid E.v.h' = E.v.h\})\ \}$ | |
| Assign to visible | $v{:=}\,E.v.h$ | $\{\ (E.v.h,\ h,\ \{h'{:}\,H \mid E.v.h' = E.v.h\})\ \}$ | $\star$ |
| Assign to hidden | $h{:=}\,E.v.h$ | $\{\ (v,\ E.v.h,\ \{h'{:}\,H \cdot E.v.h'\})\ \}$ | $\star$ |
| Choose visible | $v{:\in}\,S.v.h$ | $\{v'{:}\,S.v.h \cdot\ (v',\ h,\ \{h'{:}\,H \mid v' \in S.v.h'\})\ \}$ | $\star$ |
| Choose hidden | $h{:\in}\,S.v.h$ | $\{h'{:}\,S.v.h \cdot\ (v,\ h',\ \{h'{:}\,H; h''{:}\,S.v.h' \cdot h''\})\ \}$ | $\star$ |
| Sequential composition | $P_1; P_2$ | $([\![P_1]\!]; [\![P_2]\!]).v.h.H$ | |
| Demonic choice | $P_1 \sqcap P_2$ | $[\![P_1]\!].v.h.H \cup [\![P_2]\!].v.h.H$ | |
| Conditional | $\textbf{if}\ E.v.h\ \textbf{then}\ P_t\ \textbf{else}\ P_f\ \textbf{fi}$ | $[\![P_t]\!].v.h.\{h'{:}\,H \mid E.v.h' = \textbf{true}\}$ $\lhd E.v.h \rhd$ $[\![P_f]\!].v.h.\{h'{:}\,H \mid E.v.h' = \textbf{false}\}$ | |

The commands $P$ marked $\star$ satisfy $[\![P]\!] = \langle$ *"classical semantics of P"* $\rangle$, and we call them the *atomic* commands, meaning semantically so. Note that **reveal** is therefore not security-atomic, even though it is a syntactic atom.

**Figure 1.** Semantics of non-looping commands

## *2.5. Declared atomicity*

If there is a code fragment $P$ that we know will be executed atomically at runtime, we can write it $\langle P\rangle$ using the notation of Def. 1. This will however have two consequences:

(i) At runtime the atomicity must be guaranteed for $P$'s execution, and

(ii) At design-time only equality reasoning can be used within $P$.

With respect to (ii) we mean that $P \sqsubseteq P'$ does not allow us to conclude the refinement $\langle P\rangle \sqsubseteq \langle P'\rangle$. We can however conclude the equality $\langle P\rangle = \langle P'\rangle$ from $P{=}P'$.

## *2.6. Summary of semantics*

The Shadow Semantics of a small imperative language is given in Fig. 1 for non-looping constructs. The only non-traditional command is **reveal** that gives the value of some expression to the attacker directly, but changes no program variables; note it does change the shadow.

Refinement between programs is defined as follows:

**Definition 2.** *Refinement* For programs $P_{\{1,2\}}$ we say that $P_1$ *is refined by* $P_2$ and write $P_1 \sqsubseteq P_2$ just when for all $v, h, H$ we have

$$
\begin{aligned}
&(\forall (v', h', H_2'){:}\, [\![P_2]\!].v.h.H \cdot \\
&\qquad \big(\exists H_1'{:}\, \mathbb{P}\mathcal{H} \mid H_1' \subseteq H_2' \cdot \quad (v', h', H_1') \in [\![P_1]\!].v.h.H\big)\ )\ .
\end{aligned}
$$

This means that for each initial triple $(v, h, H)$ every final triple $(v', h', H'_2)$ produced by $P_2$ must be "justified" by a triple $(v', h', H'_1)$, with equal or smaller ignorance, produced by $P_1$. [4] $\square$

For example, from Fig. 1 we have that $[\![h{:=}\,0 \sqcap h{:=}\,1]\!].v.h.H$ produces the outome $\{(v, 0, \{0\}), (v, 1, \{1\})\}$ whereas $[\![h{:\in}\,\{0, 1\}]\!].v.h.H$ produces the outcome $\{(v, 0, \{0, 1\}), (v, 1, \{0, 1\})\}$. Thus

$$h{:=}\,0 \sqcap h{:=}\,1 \quad \sqsubset \quad h{:\in}\,\{0, 1\} \tag{3}$$

is an example of a strict refinement where the two commands differ only by a strict increase of ignorance: they have equal nondeterminism functionally, but in one case ($\sqcap$) it can be observed by the attacker and in the other case ($:\in$) it cannot. For example the "more ignorant" triple $(v, 0, \{0, 1\})$ is strictly justified by the "less ignorant" triple $(v, 0, \{0\})$, where we say "strictly" because $\{0\} \subset \{0, 1\}$.

## 3. The Logic of Ignorance

With the $(v, h, H)$-triple semantics of Sec. 2.6 comes an assertion logic over the triples; it is based on the Logic of Knowledge and its interpretation over Kripke structures [11]. We call it *The Logic of Ignorance.*

As in Hoare Logic for sequential programs [16] we interpret first-order predicate formulae over program states by making the program variables' values act as constants in the logic. To that we add a *possibility modality* so that $P\phi$ means (roughly) that $\phi$ holds for some "possible" value $h{\in}H$ rather than necessarily for the actual current value of $h$, where $\phi$ is a classical formula. In fact we have $\phi \Rightarrow P\phi$ because a property of our semantics is that $h{\in}H$ for all triples $(v, h, H)$ we consider: what is true must also be possible. In general however we do not have $P\phi \Rightarrow \phi$, since what is possible is not necessarily true.

### *3.1. Interpretation of modal formulae*

The assertion language contains function- (including constant-) and relation symbols as needed, among which we distinguish the (program-variable) constant symbols *visibles* in some set $\mathsf{V}$ and *hiddens* in $\mathsf{H}$; as well there are the usual (logical) variable symbols in $\mathsf{L}$ over which we allow $\forall, \exists$ quantification. The visibles, hiddens and variables are collectively the *scalars* $\mathsf{X}{:=}\,\mathsf{V} \cup \mathsf{H} \cup \mathsf{L}$ with $\mathsf{V}, \mathsf{H}, \mathsf{L}$ assumed disjoint.

A *structure* comprises a non-empty domain $\mathcal{D}$ of values, together with functions and relations over it that interpret the function- and relation symbols mentioned above; within the structure we name the partial functions $\mathsf{v}, \mathsf{h}$ that interpret visibles and hiddens respectively; we write their types $\mathsf{V} \nrightarrow \mathcal{D}$ and $\mathsf{H} \nrightarrow \mathcal{D}$ respectively (where the "crossbar" indicates the potential partiality of the function). We don't bother naming the interpretations of function- and relation symbols, as they do not vary from one program state to another.

[4]This is the Smyth Order [33] on sets of outcomes that is induced by the order "$(v, h, H_1) \sqsubseteq (v, h, H_2)$ iff $H_1 \subseteq H_2$" on individual outcomes.

- $(\mathsf{v},\mathsf{h},\mathsf{H}),\mathsf{w} \models R.t_1.\cdots.t_K$ for relation symbol $R$ and terms $t_{\{1\cdots K\}}$ iff the tuple $([\![t_1]\!].\mathsf{v}.\mathsf{h}.\mathsf{w},\cdots,[\![t_K]\!].\mathsf{v}.\mathsf{h}.\mathsf{w})$ is an element of the interpretation of $R$ in $\mathcal{D}^K$.
- $(\mathsf{v},\mathsf{h},\mathsf{H}),\mathsf{w} \models t_1 = t_2$ iff $[\![t_1]\!].\mathsf{v}.\mathsf{h}.\mathsf{w} = [\![t_2]\!].\mathsf{v}.\mathsf{h}.\mathsf{w}$.
- $(\mathsf{v},\mathsf{h},\mathsf{H}),\mathsf{w} \models \neg\Phi$ iff $(\mathsf{v},\mathsf{h},\mathsf{H}),\mathsf{w} \not\models \Phi$.
- $(\mathsf{v},\mathsf{h},\mathsf{H}),\mathsf{w} \models \Phi_1 \wedge \Phi_2$ iff $(\mathsf{v},\mathsf{h},\mathsf{H}),\mathsf{w} \models \Phi_1$ and $(\mathsf{v},\mathsf{h},\mathsf{H}),\mathsf{w} \models \Phi_2$.
- $(\mathsf{v},\mathsf{h},\mathsf{H}),\mathsf{w} \models (\forall x \cdot \Phi)$ iff $(\mathsf{v},\mathsf{h},\mathsf{H}),\mathsf{w}\triangleleft(x\mapsto d) \models \Phi$ for all $d$ in $\mathcal{D}$.
- $(\mathsf{v},\mathsf{h},\mathsf{H}),\mathsf{w} \models P\Phi$ iff $(\mathsf{v},\widehat{\mathsf{h}},\mathsf{H}),\mathsf{w} \models \Phi$ for some $\widehat{\mathsf{h}}$ in $\mathsf{H}$.

We write just $(\mathsf{v},\mathsf{h},\mathsf{H}) \models \Phi$ when $\mathsf{w}$ is empty, and $\models \Phi$ when $(\mathsf{v},\mathsf{h},\mathsf{H}) \models \Phi$ for all $(\mathsf{v},\mathsf{h},\mathsf{H})$ with $\mathsf{h}{\in}\mathsf{H}$, and we take advantage of the usual "syntactic sugar" for other operators. Thus for example we can show $\models \Phi \Rightarrow P\Phi$ for all $\Phi$, a fact which we mentioned earlier. Similarly we can assume *wlog* that modalities are not nested, since we can remove nestings via the validity

$$\models P\Phi \equiv \left(\exists c \cdot \Phi_{h\leftarrow c} \wedge P(h{=}c)\right) .$$

As a convenience we allow 0-subscripted hidden variables (e.g. $h_0$) within the modality to refer to the actual rather than potential hidden value; for that we extend the last clause above to read

- $(\mathsf{v},\mathsf{h},\mathsf{H}),\mathsf{w} \models P\Phi$ iff $(\mathsf{v},\widehat{\mathsf{h}},\mathsf{H}),\mathsf{w}\triangleleft(h_0\mapsto\mathsf{h}.h) \models \Phi$ for some $\widehat{\mathsf{h}}$ in $\mathsf{H}$.

Thus for example $P(h{=}\neg h_0)$ means that whatever value Boolean $h$ might have, we must consider also its negation to be possible: we do not (cannot, if that formula holds) know it exactly.

**Figure 2.** Interpretation of Logic of Ignorance

A *valuation* is a partial function from scalars to $\mathcal{D}$, thus typed $\mathsf{X} \nrightarrow \mathcal{D}$; one valuation $\mathsf{w}'$ can override another $\mathsf{w}$ so that for scalar $x$ we have that $(\mathsf{w}\triangleleft\mathsf{w}').x$ is $\mathsf{w}'.x$ if $\mathsf{w}'$ is defined at $x$ and is $\mathsf{w}.x$ otherwise. The valuation $x\mapsto d$ is defined only at variable $x$, where it takes value $d$.

A *state* $(\mathsf{v},\mathsf{h},\mathsf{H})$ comprises a visible- $\mathsf{v}$, hidden- $\mathsf{h}$ and *shadow*- part $\mathsf{H}$; the last, in $\mathbb{P}(\mathsf{H} \nrightarrow \mathcal{D})$, is a *set* of valuations over hiddens only. All the states that we consider satisfy $\mathsf{h}{\in}\mathsf{H}$.

We define truth of $\Phi$ at $(\mathsf{v},\mathsf{h},\mathsf{H})$ under valuation $\mathsf{w}$ by induction in the usual style, writing $(\mathsf{v},\mathsf{h},\mathsf{H}),\mathsf{w} \models \Phi$. For term $t$ let $[\![t]\!].\mathsf{v}.\mathsf{h}.\mathsf{w}$ be its value interpretation determined inductively from the valuation $\mathsf{v}\triangleleft\mathsf{h}\triangleleft\mathsf{w}$ and the (implicit) interpretation of function symbols. Then our formula interpretation is as defined as in Fig. 2 [11, pp. 79,81].

## *3.2. Shadow-sensitive Hoare-triples; revelations*

As is normal, we say that $\{\Phi\}\mathit{prog}\{\Psi\}$ just when any initial state $(\mathsf{v},\mathsf{h},\mathsf{H}) \models \Phi$ can lead via $[\![\mathit{prog}]\!]$ only to final states $(\mathsf{v}',\mathsf{h}',\mathsf{H}') \models \Psi$; typically $\Phi$ is called the *precondition* and $\Psi$ is called the *postcondition.*

We illustrate Shadow-sensitive Hoare-triples with the **reveal** $E$ command: we have for any classical $\phi$ that [5]

[5]We use upper case for modal formulae, and lower case for classical.

$$\{P(E{=}E_0 \wedge \phi)\}\ \textbf{reveal}\ E\ \{P\phi\}\ , \qquad \text{where } E_0 \text{ is } E \text{ with all its hidden variables 0-subscripted.} \tag{4}$$

It is verified as follows:

$$
\begin{array}{lll}
 & (\mathsf{v},\mathsf{h},\mathsf{H}) \models P(E{=}E_0 \wedge \phi) & \\
 & \text{and } [\![\textbf{reveal } E]\!].\mathsf{v}.\mathsf{h}.\mathsf{H} \ni (\mathsf{v}',\mathsf{h}',\mathsf{H}') & \\[1ex]
\text{iff} & & \text{“Fig. 2, for some } \widehat{\mathsf{h}} \in \mathsf{H} \text{ and } \mathsf{h}_0 = (h_0 \mapsto \mathsf{h}.h)\text{”} \\
 & (\mathsf{v},\widehat{\mathsf{h}},\mathsf{H}),(\mathsf{w}\triangleleft\mathsf{h}_0) \models E{=}E_0 \wedge \phi & \\
 & \text{and } [\![\textbf{reveal } E]\!].\mathsf{v}.\mathsf{h}.\mathsf{H} \ni (\mathsf{v}',\mathsf{h}',\mathsf{H}') & \\[1ex]
\text{iff} & (\mathsf{v},\widehat{\mathsf{h}},\mathsf{H}),(\mathsf{w}\triangleleft\mathsf{h}_0) \models E{=}E_0 & \text{“Fig. 2”} \\
 & \text{and } (\mathsf{v},\widehat{\mathsf{h}},\mathsf{H}),(\mathsf{w}\triangleleft\mathsf{h}_0) \models \phi & \\
 & \text{and } [\![\textbf{reveal } E]\!].\mathsf{v}.\mathsf{h}.\mathsf{H} \ni (\mathsf{v}',\mathsf{h}',\mathsf{H}') & \\[1ex]
\text{iff} & E.\mathsf{v}.\widehat{\mathsf{h}} = E.\mathsf{v}.\mathsf{h} & \text{“Fig. 2; Fig. 1”} \\
 & \text{and } (\mathsf{v},\widehat{\mathsf{h}},\mathsf{H}),(\mathsf{w}\triangleleft\mathsf{h}_0) \models \phi & \\
 & \text{and } \mathsf{v}'{=}\mathsf{v} \wedge \mathsf{h}'{=}\mathsf{h} \wedge \mathsf{H}'{=}\{\mathsf{h}'{:}\,\mathsf{H} \mid E.\mathsf{v}.\mathsf{h}' = E.\mathsf{v}.\mathsf{h}\} & \\[1ex]
\text{iff} & \widehat{\mathsf{h}} \in \mathsf{H}' & \text{“third line simplifies first; equalities”} \\
 & \text{and } (\mathsf{v}',\widehat{\mathsf{h}},\mathsf{H}),(\mathsf{w}\triangleleft\mathsf{h}_0') \models \phi & \\
 & \text{and } \mathsf{v}'{=}\mathsf{v} \wedge \mathsf{h}'{=}\mathsf{h} \wedge \mathsf{H}'{=}\{\mathsf{h}'{:}\,\mathsf{H} \mid E.\mathsf{v}.\mathsf{h}' = E.\mathsf{v}.\mathsf{h}\} & \\[1ex]
\text{iff} & \widehat{\mathsf{h}} \in \mathsf{H}' & \text{“classical } \phi \text{ has shadow-independent interpretation:} \\
 & \text{and } (\mathsf{v}',\widehat{\mathsf{h}},\mathsf{H}'),(\mathsf{w}\triangleleft\mathsf{h}_0') \models \phi & \text{thus can replace } \mathsf{H} \text{ by } \mathsf{H}'\text{”} \\
 & \text{and } \mathsf{v}'{=}\mathsf{v} \wedge \mathsf{h}'{=}\mathsf{h} \wedge \mathsf{H}'{=}\{\mathsf{h}'{:}\,\mathsf{H} \mid E.\mathsf{v}.\mathsf{h}' = E.\mathsf{v}.\mathsf{h}\} & \\[1ex]
\text{implies} & (\mathsf{v}',\widehat{\mathsf{h}},\mathsf{H}'),(\mathsf{w}\triangleleft\mathsf{h}_0') \models \phi & \text{“for some } \widehat{\mathsf{h}} \in \mathsf{H}'\text{”} \\[1ex]
\text{iff} & (\mathsf{v}',\mathsf{h}',\mathsf{H}'),\mathsf{w} \models P\phi\ . & \text{“Fig. 2”}
\end{array}
$$

That was not a pretty calculation but, having done it once, we can use (4) forever.

In fact the precondition in (4) is the *weakest* such with respect to postcondition $P\phi$, and we thoroughly explore ignorance-based weakest preconditions elsewhere [8,25,27]. Using that, we can give some examples of assertion-based reasoning about revelations.

In the items below, let $\Psi$ be the assertion $P((h\,\textbf{mod}\,2 = h_0\,\textbf{mod}\,2) \wedge h{=}3)$, generated by (4) applied to $\textbf{reveal}\ (h\,\textbf{mod}\,2)\,\{P(h{=}3)\}$ , which we can simplify as follows:

$$
\begin{array}{ll}
 & P((h\,\textbf{mod}\,2 = h_0\,\textbf{mod}\,2) \wedge h{=}3) \\
= & P((3\,\textbf{mod}\,2 = h_0\,\textbf{mod}\,2) \wedge h{=}3) \\
= & P((1 = h_0\,\textbf{mod}\,2) \wedge h{=}3) \\
= & (h\,\textbf{mod}\,2){=}1 \wedge P(h{=}3)\ ,
\end{array}
$$

that is $\textbf{odd}\,h \wedge P(h{=}3)$. With that we consider the following examples:

(i) Does $h{:}\in\{1,3\}$; **reveal** $(h\,\mathbf{mod}\,2)$ establish $P(h{=}3)$? *Yes* ✓
Command $h{:}\in\{1,3\}$ establishes both conjuncts of $\Psi$.

(ii) Does $h{:}\in\{1,5\}$; **reveal** $(h\,\mathbf{mod}\,2)$ establish $P(h{=}3)$? *No* $\times$
Command $h{:}\in\{1,5\}$ does not establish $P(h{=}3)$, which is the second conjunct of $\Psi$. Given the source code, it is obvious that $h$ cannot be 3 finally.

(iii) Does $h{:}\in\{2,3\}$; **reveal** $(h\,\mathbf{mod}\,2)$ establish $P(h{=}3)$? *No* $\times$
Command $h{:}\in\{1,5\}$ does not establish $\mathbf{odd}\,h$, which is the first conjunct of $\Psi$. One possible outcome is that 0 is revealed, which precludes $h$'s being finally 3.

(iv) Does $(h{:=}\,1 \sqcap h{:=}\,3)$; **reveal** $(h\,\mathbf{mod}\,2)$ establish $P(h{=}3)$? *No* $\times$
The left-hand command $h{:=}\,1$ –a demonic possibility which we must take into account– establishes the first conjunct of $\Psi$ but not the second. Because the nondeterminism is visible (unlike Case (i)), if the left branch is taken –and it might be– then from the source code we know that $h$ cannot be 3.

Note especially the difference between (i) and (iv). In the former, the nondeterminism occurs within an atomic command, and is therefore hidden; but, in the latter, it occurs between atomic commands, and is therefore observable.

## 4. The Algebra of Ignorance

### *4.1. Assertions: the historical motivation*

The Hoare-logic method of program correctness involves "hybrid" formulae (the triples) that are built from two formal languages: the programming language, and the assertion language. Thus $\{\phi\}\mathit{prog}\{\psi\}$ in its partial correctness interpretation holds just when every terminating execution of *prog* from an initial state satisfying $\phi$ is guaranteed to deliver a final state satisfying $\psi$.

The command **assert** $\phi$ is typically defined to act as **skip** in states satisfying $\phi$ and to "abort" (or give some error message) otherwise [35,23]. Assuming that "abort" is refined by anything, we can see that the classical program-algebraic inequality

$$\mathbf{assert}\ \phi;\ \mathit{prog} \quad\sqsubseteq\quad \mathit{prog};\ \mathbf{assert}\ \psi$$

has the same meaning as $\{\phi\}\mathit{prog}\{\psi\}$ does. It *encodes* the Hoare-triple entirely within the programming language and its in-built notion of refinement, thus within the program algebra: if $\phi$ does not hold in the initial state, then the refinement goes through because the entire left side aborts; if $\phi$ does hold, then the refinement goes though only if the right-hand side does *not* abort by delivering some final state of *prog* for which the subsequent **assert** $\psi$ aborts.

### *4.2. Revelations: the modern analogue of assertions for security*

By analogy with Sec. 3.2 we can express ignorance-logical properties of program fragments entirely within the programming language by using a special-purpose command encoding the ignorance formulae. There are two main idioms.

In the first we have

$$\textbf{assert}\ \phi;\ \textit{prog} \quad \sqsubseteq \quad \textbf{reveal}\ E;\ \textit{prog}\ , \tag{5}$$

where of course refinement is now understood in the sense of Def. 2, that is non-classically because it preserves ignorance as well as functional properties. If $\phi$ does not hold in the initial state, then the refinement goes through; otherwise, it goes through only if *prog* reveals the initial value of $E$ "anyway" (so that the explicit **reveal** $E$ on the right "does no further damage.") Thus (5) expresses "If $\phi$ holds initially, then *prog* reveals the initial value of $E$."

In the second we have

$$\textbf{assert}\ \phi;\ \textit{prog} \quad \sqsubseteq \quad \textit{prog};\ \textbf{reveal}\ E\ ,$$

expressing "If $\phi$ holds initially, then *prog* reveals the final value of $E$."

Here are some examples, in which our variables take numeric values. We have the refinement

$$\textbf{assert}\ v{\neq}0;\ v{:=}\,v{\times}h \quad \sqsubseteq \quad \textbf{reveal}\ h;\ v{:=}\,v{\times}h$$

because $h$'s initial value can be deduced by dividing $v$'s final value by its initial value, provided that initial value was not zero. The refinement does not go through without the assertion, since in the $v$-initially-zero case we cannot deduce $h$'s value. For final values we have

$$\textbf{assert}\ v{=}0;\ h{:=}\,v{\times}h \quad \sqsubseteq \quad h{:=}\,v{\times}h;\ \textbf{reveal}\ h$$

because when $v$ is zero we can see that $h$'s final value must be zero too, although in that case $h{:=}\,v{\times}h$ still tells us nothing about $h$'s initial value. The refinement does not go through without the assertion, since in the $v$-initially-nonzero case we cannot deduce $h$'s final value without knowing what its initial value was.

Further idioms are possible, for example with revelations on both sides.

## *4.3. A calculus of revelations*[6]

We now set out some of the program-algebra associated with revelations; much use of the identities will be made later.

### *4.3.1. Replacing one revelation by another*

Provided that truth of $\phi$ implies the equality $F = f(E)$ for some function $f$ depending (optionally) on other visible variables, we have

$$\textbf{assert}\ \phi;\ \textbf{reveal}\ E \quad \sqsubseteq \quad \textbf{reveal}\ F\ . \tag{6}$$

These are some examples:

| | | | |
|---|---|---|---|
| $\textbf{assert}\ h{=}0;\ \textbf{skip}$ | $\sqsubseteq$ | $\textbf{reveal}\ h$ | Here $f$ is the constant function 0. |
| $\textbf{reveal}\ h$ | $\sqsubseteq$ | $\textbf{reveal}\ h{\ominus}1$ | ... that is $h{-}1\ \textbf{max}\ 0$. |
| $\textbf{reveal}\ h{\ominus}1$ | $\not\sqsubseteq$ | $\textbf{reveal}\ h$ | Initial values 0,1 not distinguished. |
| $\textbf{assert}\ h{>}0;\ \textbf{reveal}\ h{\ominus}1$ | $\sqsubseteq$ | $\textbf{reveal}\ h$ | If $h{>}0$ then $({\ominus}1)$ is injective. |

[6]This is the title of the presentation on which the current paper is based [26].

### *4.3.2. Combining revelations*

In all cases we have

$$\textbf{reveal}\ E;\ \textbf{reveal}\ F \quad = \quad \textbf{reveal}\ (E,F)\ . \tag{7}$$

Here is an example over Booleans and exclusive-or:

$$\begin{array}{lll} & \textbf{reveal}\ x\oplus y;\ \textbf{reveal}\ y\oplus z & \text{“Write } \oplus \text{ for exclusive-or”} \\ = & \textbf{reveal}\ (x\oplus y, y\oplus z) & \text{“(7)”} \\ = & \textbf{reveal}\ (x\oplus y, x\oplus z) & \text{“(6) in both directions”} \\ = & \textbf{reveal}\ x\oplus y;\ \textbf{reveal}\ x\oplus z\ . & \text{“(7) in both directions”} \end{array}$$

### *4.3.3. Equivalence with assignment to local visible*

In all cases we have

$$\textbf{reveal}\ E \quad = \quad [\![\ \textbf{vis}\ v\cdot\ v{:=}\,E\ ]\!]\ , \tag{8}$$

highlighting the fact that scope (local vs. global) and visibility (**vis** vs. **hid**) are orthogonal: in spite of the fact that $v$ is temporary, ultimately “popped from the stack and discarded,” assigning to it while it is there does reveal the value assigned.

An example of this is given in the next section.

## *4.4. Example: specifications and the encryption lemma*

For Booleans, or isomorphically $\{0,1\}$-valued variables $x,y$ we write $x{\oplus}y{:=}\,E$ to abbreviate the specification statement $x,y{:}[x{\oplus}y = E]$ in the style of the Refinement Calculus [23,3,1], thus a command that sets $x,y$ nondeterministically to make their exclusive-or equal to $E$. We define the command to be atomic, so that $[\![x{\oplus}y{:=}\,E]\!] = \langle x,y{:}[x{\oplus}y = E]\rangle$.

A common pattern for this is $[\![\ \textbf{vis}\ v;\ \textbf{hid}\ h'\cdot\ v{\oplus}h'{:=}\,h\ ]\!]$ in the context of a declaration **hid** $h$. It is functionally equivalent to **skip** because it assigns only to local variables; we show it is Shadow-equivalent to **skip** also, i.e. that its effect of assigning to visible $v$ reveals nothing about $h$. We have

$$\begin{array}{lll} & [\![\ \textbf{vis}\ v;\ \textbf{hid}\ h'\cdot\ v{\oplus}h'{:=}\,h\ ]\!] & \\ = & [\![\ \textbf{vis}\ v;\ \textbf{hid}\ h'\cdot\ \langle v,h'{:}[v{\oplus}h' = h]\rangle\ ]\!] & \text{“defined above”} \\ = & [\![\ \textbf{vis}\ v;\ \textbf{hid}\ h'\cdot\ \langle v{:\in}\,\{0,1\};\ h'{:=}\,h{\oplus}v\rangle\ ]\!] & \text{“standard equality”}\star \\ = & [\![\ \textbf{vis}\ v;\ \textbf{hid}\ h'\cdot\ \langle v{:\in}\,\{0,1\}\rangle;\ \langle h'{:=}\,h{\oplus}v\rangle\ ]\!] & \text{“Lem. 1”} \\ = & [\![\ \textbf{vis}\ v;\ \textbf{hid}\ h'\cdot\ v{:\in}\,\{0,1\};\ h'{:=}\,h{\oplus}v\ ]\!] & \text{“Fig. 1”} \\ = & [\![\ \textbf{vis}\ v\cdot\ v{:\in}\,\{0,1\};\ [\![\ \textbf{hid}\ h'\cdot\ h'{:=}\,h{\oplus}v\ ]\!]]\!] & \text{“move scopes”} \\ = & [\![\ \textbf{vis}\ v\cdot\ v{:\in}\,\{0,1\}]\!] & \text{“assignment to local hidden is } \textbf{skip}\text{”} \\ = & \textbf{skip}\ . & \text{“assignment of visibles to local visible is } \textbf{skip}\text{”} \end{array}$$

But at $\star$ we could have written instead

$$
\begin{array}{lll}
= & [\![\ \mathbf{vis}\ v;\ \mathbf{hid}\ h' \cdot\ \langle h'{:\in}\{0,1\};\ v{:=}\,h'{\oplus}h\rangle\ ]\!] & \text{“standard equality”} \\
= & [\![\ \mathbf{hid}\ h' \cdot\ h'{:\in}\{0,1\};\ [\![\ \mathbf{vis}\ v\ \cdot\ v{:=}\,h'{\oplus}h\ ]\!]]\!] & \text{“Lem. 1; Fig. 1; move scopes”}\ \dagger \\
= & [\![\ \mathbf{hid}\ h' \cdot\ h'{:\in}\{0,1\};\ \mathbf{reveal}\ h'{\oplus}h\ ]\!]\ , & \text{“(8)”}\ \ddagger
\end{array}
$$

with both †, ‡ being interesting formulations often used in protocols: they too are therefore equal to **skip**. Each sets a hidden local Boolean $h'$ randomly and publishes its exclusive-or with some global hidden $h$; the reasoning above shows rigorously (and formally) that no information about $h$ is released by doing that.

We call that *The Encryption Lemma* and make much use of it below.

## 5. The two cryptographers[7]

Two cryptographers are about to choose from the trolley, but there are only two desserts there: a lavish cream cake, and a small biscuit. To avoid a series of insincere “after-you” exchanges, they engage in this simple protocol: a single coin is flipped privately between them; each secretly writes his dessert choice on his own napkin if the coin shows heads, or the opposite choice if it shows tails; then they hand their folded-over napkins to the waiter.

If the waiter tells them their napkin-choices differ, they can safely take the two desserts and select their actual preferences once the waiter has gone away; otherwise, to avoid embarrassment, they will forego dessert altogether.

The protocol ensures that neither cryptographer knows the other’s choice before he makes his own choice; and, whatever happens, the waiter does not find out which of them greedily chose the cream cake.[8]

Here is a Shadow-analysis of the protocol. Let the two cryptographers be $A$ and $B$ with Boolean variables $a, b$ recording whether each wants the cream cake respectively. Boolean $c$ is the shared coin. We do not model the waiter explicitly, because his function of ensuring “oblivious choices” is outside our terms of reference: we do not address the issue of possible protocol violations. The *specification* of the protocol is just

```
hid $a, b$: Bool ·
    reveal $a{\equiv}b$ ,
```

where the declarations of the hidden $a, b$ are global: we assume them in subsequent manipulations of this example.

The specification says clearly that whether $a$ and $b$ agree is to be revealed but nothing else, and in fact it is hard to think of a clearer way of saying this. And although revealing $a{\equiv}b$ reveals $a$’s value to $B$ by implication (and vice versa),

[7]While based on Chaum’s *Dining Cryptographers* [6], the story for this tiny example has been especially invented to illustrate piecewise construction of a protocol that ultimately will be quite complex. This is the smallest portion, the first step.

[8]The original story ends differently. Without a protocol, the two diners do engage in “after-you” protestations, each believing that the first choice will out of politeness have to be the small cracker; eventually however one diner just chooses the cake. Outraged, the other protests “If *I* had chosen first, I’d have taken the cracker!” “Well,” replies the first, “That’s exactly what you’ve got.”

this does not need any special treatment: it cannot be avoided, and so there is no need to mention it.[9] Thus "but nothing else" above, an informal phrase, carries the sense of "unless unavoidable."

With the declarations as given, both $a, b$ hidden, the observer is "the public" who thus cannot observe either one directly. The *implementation* under those same declarations, that is the protocol above, is derived algebraically as follows:

```
    reveal $a{\equiv}b$

=   skip;                                              "classical reasoning"
    reveal $a{\equiv}b$

=   $[\![$ hid $c$: Bool ·                     "Encryption Lemma, Sec. 4.4,
       $c{:}{\in}$ Bool;             and that (reveal $a{\equiv}b$) = (reveal $a{\oplus}b$) by (6)"
       reveal $a{\equiv}c$
    $]\!]$;
    reveal $a{\equiv}b$

=   $[\![$ hid $c$: Bool ·                                   "adjust scopes"
       $c{:}{\in}$ Bool;
       reveal $a{\equiv}c$;
       reveal $a{\equiv}b$
    $]\!]$

=   $[\![$ hid $c$: Bool ·              "Reveal Calculus, example following (7):
       $c{:}{\in}$ Bool;                   $(a{\equiv}c, a{\equiv}b)$ determines $(a{\equiv}c, b{\equiv}c)$
       reveal $a{\equiv}c$;                                     and vice versa"
       reveal $b{\equiv}c$
    $]\!]$ .
```

Note (and recall from the introduction) that our strong assumptions for the adversary mean that it is sound to model this distributed protocol with a single sequential program: adversaries' access to the individual threads is modelled by the assumption of perfect recall.

In [22, Appendix B] we illustrate some conventions for abbreviating the presentation of derivations like the one above.

## 6. The three cryptographers[10]

Three cryptographers have just had lunch, and ask for the bill. The waiter says that the bill has already been paid; and the cryptographers want to determine whether one of them paid it or whether it was paid by the NSA. In the case that

[9]We formalise this observation by observing that with the altered declarations **hid** $a$; **vis** $b$, that is $B$'s point of view, we have the equality (**reveal** $a{\equiv}b$) = (**reveal** $a{\equiv}b$; **reveal** $a$) from (6) and $b$'s being visible.

[10]Three diners is Chaum's example exactly.

one of them paid, none of the other cryptographers nor anyone else is to be able to determine which one it was. They proceed as follows.

They are sitting at a round table,[11] and each of the three adjacent pairs flips a coin between them that only that pair can see; thus each cryptographer can see two coins, because he is a member of two such pairs.

Each cryptographer then announces whether he paid; but if the two coins he sees show different faces, he lies. If an odd number of cryptographers claim to have paid, then indeed one did, but no-one (except him) knows who it was; otherwise the lunch was paid for by the NSA.

## *6.1. Helping three cryptographers by considering one at a time*

Rather than give a direct derivation in the style of Sec. 5, we build this protocol up from smaller components. We imagine a single cryptographer $X$ with Boolean $x$ who has access to two coins $l, r$ on his left and right. The left one is already flipped; the right one he must flip himself; and then he reveals the exclusive-or of all three values. That amounts to the fragment

$$\begin{array}{l} \textbf{var}\ l, r\colon \textbf{Bool};\ \ \textbf{hid}\ x\colon \textbf{Bool}\ \cdot \\ \quad \left.\begin{array}{l} r{:}{\in}\ \textbf{Bool}; \\ \textbf{reveal}\ l \oplus x \oplus r\ , \end{array}\right\}\ \text{Protocol } X \end{array}$$

in which for the moment we are not giving the visibility type of $l, r$.

Now if we instantiate the $X$-fragment to $A$ and $B$ in turn, and introduce a hidden "middle" coin $m\colon \textbf{Bool}$, with both fragments we can get

$$\begin{array}{l} \textbf{var}\ l, r\colon \textbf{Bool};\ \ \textbf{hid}\ a, b\colon \textbf{Bool}\ \cdot \\ \quad [\![\ \textbf{hid}\ m\colon \textbf{Bool}; \\ \qquad \left.\begin{array}{l} m{:}{\in}\ \textbf{Bool}; \\ \textbf{reveal}\ l \oplus a \oplus m; \end{array}\right\}\ \text{First instance of } X \\ \qquad \left.\begin{array}{l} r{:}{\in}\ \textbf{Bool}; \\ \textbf{reveal}\ m \oplus b \oplus r \end{array}\right\}\ \text{Second instance of } X \\ \quad ]\!]\ , \end{array} \tag{9}$$

and this –by similar reasoning to Sec. 5's– can be shown[12] to implement the specification

$$\begin{array}{l} \textbf{var}\ l, r\colon \textbf{Bool};\ \ \textbf{hid}\ a, b\colon \textbf{Bool}\ \cdot \\ \quad r{:}{\in}\ \textbf{Bool}; \\ \quad \textbf{reveal}\ l \oplus (a \oplus b) \oplus r\ . \end{array} \tag{10}$$

[11]This Arthurian concept is one of Formal Methods' great contributions to computing.

[12]Think of $A$'s secret in Sec. 5 being $l \oplus a$ and $B$'s secret being $b \oplus r$, and re-instantiate the derivation on that basis, replacing $\equiv$ by $\oplus$.

Again only $a \oplus b$ is revealed, and nothing about $a$ or $b$ individually. But the point of doing it in this way is that it suggests how the protocol can be extended to any number of participants. So far we have dealt with two out of three.

For the third cryptographer (or final, when there are more than three in total) we must use a slightly different approach. It is no more complex, but must be "backwards" since he cannot assume that some coin is already flipped: the process must begin somewhere; and the two "extremal" coins must be hidden. Thus Cryptographer $C$ executes

$$\begin{array}{ll}
[\![\ \textbf{hid}\ l, r\colon \textbf{Bool}\ \cdot & \\
\quad l{:\in}\ \textbf{Bool}; & \\
\quad \left.\begin{array}{l} r{:\in}\ \textbf{Bool}; \\ \textbf{reveal}\ l \oplus (a \oplus b) \oplus r \end{array}\right\} & \begin{array}{l}\text{specification from (10) above} \\ \text{implemented by (9) above}\end{array} \qquad (11) \\
\quad \textbf{reveal}\ l \oplus c \oplus r & \\
]\!]\ , &
\end{array}$$

in which we have embedded the *specification* of the $A, B$ protocol as the middle two commands. That is, Cryptographer $C$ flips a coin $l$ and says to $A, B$ "now execute your protocol," finally making his own revelation using the coin $l$ he flipped himself (now some time ago) and the "output" coin $r$ provided by the $A, B$ protocol he arranged to have executed. This is the right thing to do, because in two easy steps from the above we can reason

$$\begin{array}{lll}
(11) & = & [\![\ \textbf{hid}\ l, r\colon \textbf{Bool}\ \cdot \qquad \text{"Revelation Calculus; adjust scopes"} \\
& & \quad l{:\in}\ \textbf{Bool}; \\
& & \quad r{:\in}\ \textbf{Bool}; \\
& & \quad \textbf{reveal}\ l \oplus (a \oplus b) \oplus r \\
& & ]\!]; \\
& & \textbf{reveal}\ a \oplus b \oplus c \\
\\
& = & \textbf{reveal}\ a \oplus b \oplus c\ , \qquad \text{"Encryption Lemma for } l, r \text{ together"}
\end{array}$$

which is our specification for the Three Cryptographers.

To finish the three cryptographers' protocol we now simply replace the specification of $A, B$'s sub-protocol by its implementation, which was given earlier. Because the monotonicity property of refinement, actually equality in this case, we do not need to do any further checking. The immediate result, thus obtained "for free" from (10) $\sqsubseteq$ (9), is

$$\begin{array}{l}
\textbf{hid}\ a, b, c\colon \textbf{Bool}\ \cdot \\
\\
\quad \textbf{reveal}\ a \oplus b \oplus c
\end{array}$$

```
=  ⟦ hid $l, m, r$: Bool ·                    "replace $A, B$ specification above
     $l$:∈ Bool;                               by its implementation from earlier"
     $m$:∈ Bool;
     reveal $l \oplus a \oplus m$;
     $r$:∈ Bool;
     reveal $m \oplus b \oplus r$;
     reveal $l \oplus c \oplus r$
   ⟧ .
```

In Sec. 8 we will do the same step-by-step construction within a loop, thus dealing with arbitrarily many cryptographers.

## *6.2. On expressiveness and "caveats"*

An informal specification of the Three Cryptographers might state that whether the NSA paid is to be learned without at the same time learning whether any particular cryptographer paid. Except of course that cryptographer himself, who knows it anyway... Similarly, as we saw, it is unavoidable that each of the Two Cryptographers learns what the other chose, given that he knows his own choice and comes to know whether the other's differs.

Thus if the first sentence above were formalised, as a logical assertion to be met by the implemented code, it would be too strong. The *caveat* (A) is that when we write (somehow) "for all $i, j$: 1..3 cryptographer$_i$ does not know whether cryptographer$_j$ paid," we must add (when we remember) "provided $i \neq j$."

Similarly there is an implicit assumption that at most one cryptographer paid (where "implicit" means "probably we forgot to mention that the first time around"). If two cryptographers paid (B), then the outcome will be "NSA paid" when in fact it did not: two of the three cryptographers did. So another caveat is added: "Assuming that at most one cryptographer paid..."

In fact neither of these two problems bother us if we use refinement. In both cases (A,B) it is obvious from the *specification* **reveal** $a \oplus b \oplus c$ what behaviour we should expect in all situations, no matter how bizarre, and we do not have to add extra "caveat" clauses to some assertion in order to accommodate them. More importantly, we do not have to worry about whether we have added *enough* caveat clauses. A similar situation occurs in the *Obvlivious Transfer Protocol* [29,10,30], specified $a{:=}\,b_i$ and in which $A$ reads into $a$ his choice indexed $i$: $\{1, 2\}$ of one of two messages $b_{1,2}$ that $B$ holds, without $A$'s learning anything about the message he did not choose and without $B$'s learning anything about the index $i$ of the choice $A$ made. Except that in the case $m_1{=}m_2$ we must accept (C) that $A$ does learn about the message he did not choose, because it is equal to the one he did choose...

Again, from the specification $a{:=}\,b_i$ it is obvious what happens in (C), and we do not have to introduce caveats to accommodate it. (We gave a rigorous derivation of the Oblivious Transfer Protocol in our earlier report [27].)

## *6.3. On points of view*

In the derivation of Sec. 6.1 all three variables $a, b, c$ are declared hidden, and so our conclusions apply only to adversaries for whom they actually *are* all-three hidden: the general public. To show that as well that no cryptographer learns the thoughts of another, say that $C$ does not learn about whether $A$ or $B$ paid (unless of course $C$ did pay, in which case he knows that $A$ and $B$ did not... another caveat we can ignore), we would vary the declarations **hid** $a, b$; **vis** $c$ and do the derivation under those conditions.

In general, sometimes the same derivation steps go through for all viewpoints; but sometimes they do not, and then we must choose different intermediate refinement steps depending on "who's looking." When that happens, it's equivalent to a case analysis and can fairly be considered a disadvantage: thus we try to find derivations that go through for all viewpoints in the same way.

## 7. Loops and fixed-points

As an example of how loops are treated, the code of (1) in Sec. 1, slightly modified, is shown to satisfy a simple specification: we will prove the equality

```
    hid $h{:}\,\mathbb{N}$ ·
       reveal $h{\div}2$;
       $h{:=}\,h$ mod $2$

=   while $h{>}1$ do
       $h{:=}\,h-2$
    end .
```

That is, not only does the loop change the value of $h$ (in an obvious way), but the repeated conditional tests reveal all but the low bit of $h$'s original value. This leaking occurs because it is a refinement (an equality) to unfold a loop, which produces an **if** command, and we have already seen how refinement causes leakage in the conditionals of **if**'s.

Terminating loops are the unique fixed-points of their associated program functionals, and so to prove equality between a loop and some specification it is enough to show the specification satisfies the loop's functional. In the example above, that means we show

```
    hid $h{:}\,\mathbb{N}$ ·
       reveal $h{\div}2$;
       $h{:=}\,h$ mod $2$

=   if $h{>}1$ then
       $h{:=}\,h-2$;
       reveal $h{\div}2$;
       $h{:=}\,h$ mod $2$
    fi ,
```

for which the techniques we have already will suffice.

We start with the right-hand side, since it has more structure (thus suggesting appropriate moves), and the left-hand side is a smaller target:

```
    if $h>1$ then
        $h:= h - 2$;
        reveal $h \div 2$;
        $h:= h$ mod 2
    fi

=   if $h>1$ then                                          “add assertion”
        assert $h>1$;
        $h:= h-2$;
        reveal $h \div 2$;
        $h:= h$ mod 2
    fi

=   if $h>1$ then                                          “commute commands”
        assert $h>1$;
        reveal $(h-2) \div 2$;
        $h:= h-2$;
        $h:= h$ mod 2
    fi

=   if $h>1$ then        “Revelation calculus; classical reasoning; remove assertion”
        reveal $h \div 2$;
        $h:= h$ mod 2
    fi

=   if $h>1$ then          “Add assertion; Revelation Calculus; classical reasoning”
        reveal $h \div 2$;
        $h:= h$ mod 2
    else
        assert $0 \le h \le 1$;
        reveal $h \div 2$;
        $h:= h$ mod 2
    fi

=   if $h>1$ then skip else skip fi;       “Remove assertion; classical reasoning”
    reveal $h \div 2$;
    $h:= h$ mod 2

=   reveal $h>1$;                                          “Revelation calculus”
    reveal $h \div 2$;
    $h:= h$ mod 2

=   reveal $h \div 2$;                                     “Revelation calculus”
    $h:= h$ mod 2 ,
```

and we are done.

An abbreviated derivation is given in [22, Appendix C].

## 8. The thousand-and-one cryptographers[13]

With the tools introduced in earlier sections, we can now derive a looping program that implements the Dining Cryptographers' specification for as many participants as we like. Let the cryptographers be numbered $0..N$ inclusive (thus $N{+}1$ of them) and let their did-pay states be recorded as indexed Boolean variables $a[0..N]$: **Bool**. Our specification is then

```
vis $N{:}\,\mathbb{N}$; hid $a[0..N]{:}\,\mathbf{Bool}$ ·

  reveal $(\oplus n{:}\,\mathbb{N} \mid 0{\le}n{\le}N \cdot a[n])$ ,
```

Having learned in Sec. 6.1 that the last cryptographer is treated specially, we make that special treatment our first development step, reasoning

```
=    ⟦ hid $l, r{:}\,\mathbf{Bool}$ ·                                  "As in Sec. 6.1"
       reveal $l \oplus (\oplus n{:}\,\mathbb{N} \mid 0{\le}n{<}N \cdot a[n]) \oplus r$;   |
       reveal $l \oplus a[N] \oplus r$
     ⟧ ,
```

intending to implement the right-barred portion (i.e. having a "|" at right) as a loop.

For that loop, we refer again to Sec. 6.1, which suggests using a loop body built on the fragment

```
$r{:}{\in}\,\mathbf{Bool}$;
reveal $l \oplus a[n] \oplus r$;
$n{:=}\,n{+}1$ ,
```

and it turns out that a **repeat-until** works better in this instance. With that in mind we propose as the next step for the right-barred portion, above, the code

```
=    ⟦ vis $n{:}\,\mathbb{N}$; hid $m$ ·
       $m, n{:=}\,l, 0$;
       repeat                          |
         $r{:}{\in}\,\mathbf{Bool}$;            |
         reveal $m \oplus a[n] \oplus r$;  |
         $m, n{:=}\,r, n{+}1$           |
       until $n{=}N$                   |
     ⟧ ,
```

where we have had to introduce a temporary variable $m$ to avoid over-writing the initially flipped $l$ that will be needed at the end by Cryptographer $N$. In order to establish this equality, we use the techniques of Sec. 7 to show that the right-barred **repeat-until** is equal to this straight-line fragment:

[13]This is in the Arabian sense: "as many as you like."

```
    vis $N{:}\,\mathbb{N}$;  hid $a[0..N]{:}\,$Bool $\cdot$

       reveal $(\oplus n{:}\,\mathbb{N} \mid 0{\le}n{\le}N \cdot a[n])$

=   $[\![$ vis $n{:}\,\mathbb{N}$;  hid $l,m,r{:}\,$Bool $\cdot$              "Reasoning in this section"
       $l{:}{\in}\,$Bool;
       $m,n{:}{=}\,l,0$;
       repeat
          $r{:}{\in}\,$Bool;
          reveal $m \oplus a[n] \oplus r$;
          $m,n{:}{=}\,r,n{+}1$
       until $n{=}N$;
       reveal $l \oplus a[N] \oplus r$
    $]\!]$
```

**Figure 3.** Specification and implementation for the thousand-and-one cryptographers.

```
    $r{:}{\in}\,$Bool;
    reveal $m \oplus (\oplus i{:}\,\mathbb{N} \mid n{\le}i{<}N \cdot a[i]) \oplus r$
    $m,n{:}{=}\,r,N$
```

For the loop (and its functional) as given, that means we must work towards the program fragment immediately above from this fragment below:

```
    $r{:}{\in}\,$Bool;
    reveal $m \oplus a[n] \oplus r$;
    $m,n{:}{=}\,r,n{+}1$;

    if $n{<}N$ then
       $r{:}{\in}\,$Bool;
       reveal $m \oplus (\oplus i{:}\,\mathbb{N} \mid n{\le}i{<}N \cdot a[i]) \oplus r$;
       $m,n{:}{=}\,r,N$
    fi .
```

Since this derivation is "more of the same" material that we have illustrated in earlier sections.

As we remarked in Sec. 6.1, monotonicity of refinement (equivalently, the congruence of our program operators) means that no further reasoning is necessary when we pull the pieces of this section together. That gives the overall equality shown in Fig. 3.

## 9. Advantages; disadvantages; comparisons; conclusions

Provable program refinement is the established scientific technique relating specifications to software code; it is hard to achieve, but brings with it a recognised quality to the workmanship of the code it produces. "Provable security refinement" –or something very like– is the technique we propose here with similar implications to quality.

Our proposed mathematical model for secure refinement has has been inspired by a number of other works; our contribution has been to select and fuse several well-known techniques to produce a reasoning tool that can be applied at the source level, and our focus on reasoning *at the level of source code* is the most obvious feature setting us apart from other researchers. Earlier work setting out the theory [25,26] outlined in more detail how this approach relates to other techniques. In summary it shares many similarities with the Logic of Knowledge [15] but is less general. The semantic technique is based on a version of noninterference which distinguishes "high-security" variables from "low security", and similar techniques have been suggested by Leino [19] and Sabelfeld [32].

However our overriding motivation is to be able to prove security properties about program code relative to specific assumptions about the operating context. But code –even without security implications– is hard to understand; with security in the mix it can rise to a higher order of impenetrability, and finding security flaws in such code is an unending task. In 1988 Goldwasser, Micali and Rivest [13] were the first to introduce the idea of "provable security"; it was highly innovative for its time but set the foundations to place security on a scientific footing, and has led to many theoretical results about cryptographic protocols and their relationship to their underlying cryptographic primitives. Although we do not claim a technique as general or widely applicable as Goldwasser and Micali's work, we do claim a source level method following its fundamental principals which is applicable to some security properties. Here our attacker –a feature of their work– is the programmer who might (maliciously or not) attempt to use a program in a context for which it was not designed; secure refinement means exactly that the implemented code has the same (or better) security properties as the specification. The crucial advantage of this is that the specification suffers exactly the same security flaws as the implementation, whatever they might be, and is exposed to *the same attacks*. Specifications by tradition only state the designer's ideal requirements and avoid the issues of implementation, and –as with traditional functional properties– it is only at the abstract level that designers have any chance of understanding their designs: this is where security issues should be considered.

## *Acknowlegements*

Reprinted/adapted by permission from Springer Nature: *The Thousand-and-one Cryptographers* by Annabelle McIver and Carroll Morgan in *Reflections on the work of C.A.R. Hoare* (C.B. Jones and A.W. Roscoe (editors)), Springer, 2010.

## References

[1] J.-R. Abrial. *The B Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[2] Rajeev Alur, Pavol Černý, and Steve Zdancewic. Preserving secrecy under refinement. In *ICALP '06: Proceedings (Part II) of the 33rd International Colloquium on Automata, Languages and Programming*, pages 107–118. Springer, 2006.

[3] R.-J.R. Back. On the correctness of refinement steps in program development. Report A-1978-4, Dept Comp Sci, Univ Helsinki, 1978.

[4] R.-J.R. Back. A calculus of refinements for program derivations. *Acta Inf*, 25:593–624, 1988.

[5] Philippa J. Broadfoot and A. W. Roscoe. Tutorial on FDR and its applications. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN*, volume 1885 of *Lecture Notes in Computer Science*, page 322. Springer, 2000.

[6] D. Chaum. The Dining Cryptographers problem: Unconditional sender and recipient untraceability. *J. Cryptol.*, 1(1):65–75, 1988.

[7] Stephen Chong and Andrew C. Myers. Security policies for downgrading. In *CCS '04: Proceedings of the 11th ACM Conference on Computer and Communications Security*, pages 198–209, New York, NY, USA, 2004. ACM.

[8] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[9] K. Engelhardt, Y. Moses, and R. van der Meyden. Unpublished report, Univ NSW, 2005.

[10] Shimon Even, Oded Goldreich, and Abraham Lempel. A randomized protocol for signing contracts. *Commun. ACM*, 28(6):637–647, 1985.

[11] R. Fagin, J. Halpern, Y. Moses, and M. Vardi. *Reasoning about Knowledge.* MIT Press, 1995.

[12] J.A. Goguen and J. Meseguer. Unwinding and inference control. In *Proc IEEE Symp on Security and Privacy*, pages 75–86, 1984.

[13] S. Goldwasser, S. Micali, and R. Rivest. A digital signature scheme secure against adaptive chosen message attacks. *SIAM J. on Computing*, 17:281–308, 1988.

[14] Probabilistic Systems Group. Collected publications. `www.cse.unsw.edu.au/~carrollm/probs`.

[15] J.Y. Halpern and K.R. O'Neill. Anonymity and information hiding in multiagent systems. In *Proc 16th IEEE Computer Security Foundations Workshop*, pages 75–88, 2003.

[16] C.A.R. Hoare. An axiomatic basis for computer programming. *Comm ACM*, 12(10):576–80, 583, October 1969.

[17] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall International, 1985.

[18] J. Jacob. Security specifications. In *IEEE Symposium on Security and Privacy*, pages 14–23, 1988.

[19] K.R.M. Leino and R. Joshi. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1–3):113–38, 2000.

[20] Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In *POPL '05: Proc. 32nd ACM SIGPLAN-SIGACT Symp. on Princ. of Prog. Lang.*, pages 158–170, New York, NY, USA, 2005. ACM.

[21] Heiko Mantel. Preserving information flow properties under refinement. In *Proc IEEE Symp Security and Privacy*, pages 78–91, 2001.

[22] A.K. McIver and C.C. Morgan. The thousand-and-one cryptographers. In K. Wood A. Roscoe, C. Jones, editor, *Reflections on the Work of C.A.R. Hoare*, pages 255–282. Springer, 2010.

[23] C.C. Morgan. *Programming from Specifications.* Prentice-Hall, second edition, 1994. `web.comlab.ox.ac.uk/oucl/publications/books/PfS/`.

[24] C.C. Morgan. Of probabilistic *wp* and *CSP*. In A. Abdallah, C.B. Jones, and J.W. Sanders, editors, *Communicating Sequential Processes: The First 25 Years*. Springer, 2005.

[25] C.C. Morgan. *The Shadow Knows:* Refinement of ignorance in sequential programs. In T. Uustalu, editor, *Math Prog Construction*, volume 4014 of *Springer*, pages 359–78. Springer, 2006. Treats *Dining Cryptographers.*

[26] C.C. Morgan. A calculus of revelations, 2008. Presented at *VSTTE '08*, Toronto. `http://www.cs.stevens.edu/~naumann/vstte-theory-2008/`.

[27] C.C. Morgan. *The Shadow Knows:* Refinement of ignorance in sequential programs. *Science of Computer Programming*, 74(8), 2009. Treats *Oblivious Transfer.*

[28] C.C. Morgan and A.K. McIver. Unifying *wp* and *wlp*. *Inf Proc Lett*, 20(3):159–64, 1996. Available at [14, key MM95].

[29] Michael O. Rabin. How to exchange secrets by oblivious transfer. Technical Report TR-81, Harvard University, 1981. Available at `eprint.iacr.org/2005/187`.

[30] R. Rivest. Unconditionally secure commitment and oblivious transfer schemes using private channels and a trusted initialiser. Technical report, M.I.T., 1999. `//theory.lcs.mit.edu/~rivest/Rivest-commitment.pdf`.

[31] A.W. Roscoe, J.C.P. Woodcock, and L. Wulf. Non-interference through determinism. *Journal of Computer Security*, 4(1):27–54, 1996.

[32] A. Sabelfeld and D. Sands. A PER model of secure information flow. *Higher-Order and Symbolic Computation*, 14(1):59–91, 2001.

[33] M.B. Smyth. Power domains. *Jnl Comp Sys Sci*, 16:23–36, 1978.

[34] Niklaus Wirth. Program development by stepwise refinement. *Comm ACM*, 14(4):221–7, 1971.

[35] Niklaus Wirth and C. A. R. Hoare. A contribution to the development of ALGOL. *Communications of the ACM*, 9(6):413–432, June 1966.

*Engineering Secure and Dependable Software Systems*
*A. Pretschner et al. (Eds.)*
*IOS Press, 2019*
*© 2019 The authors and IOS Press. All rights reserved.*
*doi:10.3233/978-1-61499-977-5-163*

# Maude-NPA and Formal Analysis of Protocols with Equational Theories

Catherine MEADOWS [a,1], Santiago ESCOBAR [b] and José MESEGUER [c]
[a] *Naval Research Laboratory, Washington DC, US*
[b] *Universitat Politècnica de València, Valencia, Spain*
[c] *University of Illinois at Urbana-Champaign, Urbana, IL, US*

**Abstract.** This paper describes the latest version (3.x) of the Maude-NRL Protocol Analyzer (Maude-NPA), a tool for the analysis of cryptographic protocols that provides support for equational theories. A key feature is it supports reasoning about a class of theories with disjoint compositions that have the *finite variant property*, via the use of a unification algorithm called *variant unification*. These theories include exclusive-or, Abelian groups, and theories underlying Diffie-Hellman key exchange. In this paper we show how Maude-NPA works, and we explain in detail the variant unification algorithm it uses and how the tool makes use of it. We also describe work on another form of unification, *asymmetric unification*, that has the potential to improve on the performance of variant unification in some cases.

**Keywords.** cryptographic protocols, formal methods, rewriting, unification

## 1. Introduction

Maude-NPA is a tool for analyzing cryptographic protocols in the Dolev-Yao model. In such a model, the functions used in the protocol are expressed as terms in a *term algebra*, and communication between honest principals takes place via a network controlled by an intruder who can read, redirect, and alter traffic, create messages on its own, and may also possess the capabilities (e.g., cryptographic function and keys) available to legitimate users of the system. The terms in general behave like black boxes: the intruder cannot obtain a term $m$ unless it either receives $m$ or it can derive it from the terms it has already received using a set of *derivation rules*. For example, if the intruder obtains $k$ and $e(k,m)$, where $k$ is a key and $e$ is an encryption function, it can obtain $m$ via the derivation rule $\{K,e(K,M)\} \vdash M$ where $K$ and $M$ are variables standing for keys and messages, respectively.

This term algebra model appears to be crude, but it has actually been used quite successfully in discovering subtle flaws in protocols. Nevertheless, it is often the case that it is necessary to represent properties of the cryptosystem in a more concrete way, as equational theories obeyed by the term algebra. For example, one could have two binary operators, $e$ for decryption, and $d$ for decryption, related by the *simplification*

[1]Corresponding Author: Center for High Assurance Computer Systems, Naval Research Laboratory, Washington, DC, 20375; E-mail: catherine.meadows@nrl.navy.mil

*rule* $d(K, e(K, M)) \to M$. Then, whenever $d(k, e(k, m))$ appears in a message, it can be replaced by $m$. Such a representation is more expressive than the simple derivation rule, since it allows us to represent the application of the decryption function to a term that may not stand for an encrypted message, e.g. $d(k, m)$.

The need for equational theories becomes even more pressing, once associative-commutative theories are introduced. Since many cryptographic algorithms are based on the use of Abelian groups, this is hard to avoid. For example, in the Diffie-Hellman key exchange protocol, Alice sends to Bob the term $g^{N_A}$ modulo a prime $P$, where $N_A$ is a random nonce and $g$ is a generator of the multiplicative group of integers modulo $P$. Likewise, Bob sends $g^{N_B}$ to Alice. They, then, compute $(g^{N_B})^{N_A}$ and $(g^{N_A})^{N_B}$, respectively. We then have $(g^{N_B})^{N_A} = g^{N_B * N_A}$ and $(g^{N_A})^{N_B} = g^{N_A * N_B}$, respectively, where $*$ is an associative-commutative operator. Thus, $A$ and $B$ share the same term, which cannot be learned by a passive observer, since that would require knowledge of $N_A$ or $N_B$. However, it will not be useful to express commutativity $X * Y = Y * X$ as a rewrite rule: both sides of the equation are identical, and the rewrite rule will be applied forever. It is thus necessary to treat associative-commutative properties differently, not as *rules* but as *axioms*.

In this paper we show how such equational theories are handled in Maude-NPA, while pointing out the relevance to other cryptographic protocol analysis tools as well. We first give an overview in Section 2 of how Maude-NPA works, and how it uses *equational unification*, i.e., solving equations modulo equational theories, to express transition from one state to another. We use this to motivate the use of *variant unification* in Section 3, a general unification method that can be applied to theories that can be expressed as the disjoint union of a set of rewrite rules $E$ and some subset $B$ of associative and/or commutative and/or identity axioms. In Section 4, we present a case study of a cryptographic API described in Maude-NPA that involves a number of complex features, all handled by variant unification. We then give in Section 5 an evaluation of variant unification with respect to five desiderata for unification in cryptographic protocol analysis, pointing out its advantages and deficiencies. Finally, we describe in Section 6 research on a form of unification, called asymmetric unification, that addresses some of the deficiencies of variant unification. We conclude in Section 7.

Much of the work presented in this paper has appeared elsewhere, but has been scattered among different papers and technical reports, and required a background in rewriting and equational unification to be understood. In this paper we bring discussion of research issues together with an introduction to variant unification presented in the Maude-NPA 3.X manual intended for the general reader. In particular, most of Section 2 and a large part of Section 3 are adapted from the Maude-NPA manual, while Section 4 summarizes the work of González-Burgueño et al. in [1], and Section 6 summarizes the work of Erbatur et al. in [2].

## 2. How Maude-NPA works

In this section we give an overview of how Maude-NPA works. We do not attempt to give all the details; in particular our description of Maude-NPA states leaves out certain features that are irrelevant to the discussion. Our goal rather is to provide enough detail to motivate the use of variant unification. On the other hand, we go into great detail about how terms are constructed in Maude-NPA, since this is crucial to the understanding how variant unification works.

## 2.1. Signatures and Term Algebras

Here we give some basic definitions concerning signatures and term algebras that are necessary to understand the next section.

In Maude-NPA, a protocol $\mathscr{P}$ is specified using an order-sorted signature defined by the user, that declares the sorts and function symbols used in the protocol. An *order-sorted signature* is a pair $((\mathsf{S},<),\mathscr{F})$, where $(\mathsf{S},<)$ is a partially ordered set of types called *sorts*, and $<$ is the *subsort inclusion order* (for example, we may have $\mathsf{Nat},\mathsf{Int},\mathsf{Rat} \in \mathsf{S}$ and $\mathsf{Nat} < \mathsf{Int} < \mathsf{Rat}$), and $\mathscr{F}$ is a set of *typed function symbols* of the form $f : \mathsf{s_1} \cdots \mathsf{s_n} \to \mathsf{s}$, where $n \geq 0$, and $\mathsf{s_1},\ldots,\mathsf{s_n},\mathsf{s} \in \mathsf{S}$ (for example, we may have $0 :\to \mathsf{Nat}$, $+ : \mathsf{Nat}\ \mathsf{Nat} \to \mathsf{Nat}$, and $+ : \mathsf{Int}\ \mathsf{Int} \to \mathsf{Int}$ in $\mathscr{F}$). Consider now a family $X = \{X_\mathsf{s}\}_{\mathsf{s}\in\mathsf{S}}$ of S-sorted *variables*, where for each sort $\mathsf{s} \in \mathsf{S}$, $X_\mathsf{s}$ is an infinite set of variables of sort s (for example, for sort Nat, we may have $X_{\mathsf{Nat}} = \{x_0{:}\mathsf{Nat}, x_1{:}\mathsf{Nat}, \ldots, x_n{:}\mathsf{Nat}, \ldots\}$, and likewise for other sorts). Then, the *term algebra* $T_\Sigma(X)$ is defined inductively as an S-sorted family $T_\Sigma(X) = \{T_\Sigma(X)_\mathsf{s}\}_{\mathsf{s}\in\mathsf{S}}$ together with an *interpretation* in $T_\Sigma(X)$ of the operators in $\mathscr{F}$ as follows: (i) if $x \in X_\mathsf{s}$, then $x \in T_\Sigma(X)_\mathsf{s}$, (ii) if $a :\to \mathsf{s}$ in $\mathscr{F}$, then $a \in T_\Sigma(X)_\mathsf{s}$, (iii) if $f : \mathsf{s_1} \cdots \mathsf{s_n} \to \mathsf{s}$ in $\mathscr{F}$ and $t_i \in T_\Sigma(X)_{\mathsf{s_i}}$, $1 \leq i \leq n$, then $f(t_1,\ldots,t_n) \in T_\Sigma(X)_\mathsf{s}$, and (iv) if $t \in T_\Sigma(X)_\mathsf{s}$ and $\mathsf{s} < \mathsf{s}'$, then $t \in T_\Sigma(X)_{\mathsf{s}'}$. The *operations* of $\mathscr{F}$ are *interpreted* in $T_\Sigma(X)$ as expected: each $f : \mathsf{s_1} \cdots \mathsf{s_n} \to \mathsf{s}$ in $\mathscr{F}$ is interpreted as the lambda expression $\lambda(x_1,\ldots,x_n) \in T_\Sigma(X)_{\mathsf{s_1}} \times \cdots \times T_\Sigma(X)_{\mathsf{s_n}}\ .\ f(x_1,\ldots,x_n) \in T_\Sigma(X)_\mathsf{s}$.

For example, consider a signature with sorts Msg, Encryption, Concatenation, Nonce, Fresh, and Name. The order-sorted information is provided as a subsort inclusion order between sorts: $\mathsf{Encryption},\mathsf{Concatenation},\mathsf{Name} < \mathsf{Msg}$ describing that, for example, any term of sort Concatenation is also of sort Msg.

In this signature, we may have operations such as *pk* (for "public key encryption") with typing $pk : \mathsf{Name}\ \mathsf{Msg} \to \mathsf{Msg}$, *sk* (for "secret key encryption") with typing $sk : \mathsf{Name}\ \mathsf{Msg} \to \mathsf{Msg}$, *n* (for "nonce") with typing $n : \mathsf{Name}\ \mathsf{Fresh} \to \mathsf{Nonce}$, and ; (for "concatenation") with typing $\_;\_ : \mathsf{Msg}\ \mathsf{Msg} \to \mathsf{Msg}$ (where we use underbars to indicate argument places, i.e., $m;m'$ is the concatenation of $m$ and $m'$).

For example, the term $t = n(a,r)\ ;\ (X\ ;\ n(b,r'))$, where $a$, $b$, and $c$ are constants of sort Name, $X$ is a variable of sort Msg, and $r$, $r'$, and $r''$ are variables of sort Fresh, is a term of sort Concatenation, and hence is also a term of sort Msg.

It is also possible to specify the *algebraic properties* of the function symbols in $\Sigma$ using a set $E$ of equations. Then the pair $(\Sigma,E)$ is called an *equational theory*. This is useful for describing both the properties of the cryptographic functions in the protocol and the properties of any other symbol. For example, the symbols pk and sk satisfy the cancellation property, i.e., $sk(A,pk(A,M)) = M$ where $A$ is a variable of sort Name and $M$ is a variable of sort Msg.

Maude-NPA works with $E$-equivalence classes. For a term $t$, $[t]$ denotes its equivalence class modulo $E$ (i.e., $t' \in [t] \Leftrightarrow t' =_E t$), where $t' =_E t$, means that $t'$ may be transformed into $t$ by a sequence of elementary equality steps using equations in $E$ (i.e., replacing equals by equals). For example, if we assume that the symbol $\_;\_$ is associative, i.e., $x;(y;z) = (x;y);z$, the equivalence class $[t]$, for $t$ the example term mentioned above, contains not only $n(a,r)\ ;\ (X\ ;\ n(b,r'))$ (i.e., first compute $(X\ ;\ n(b,r'))$ and then compute $n(a,r)\ ;\ (X\ ;\ n(b,r')))$ but also $(n(a,r)\ ;\ X)\ ;\ n(b,r')$ (i.e., first compute $(n(a,r)\ ;\ X)$ and then compute $(n(a,r)\ ;\ X)\ ;\ n(b,r'))$. If we further assume that the symbol $\_;\_$ is associative and commutative, i.e., we also have $x;y = y;x$, then the elements in the equivalence class $[t]$ include all the terms obtainable by permuting the subterms immediately below the symbol occurrences of ; in $n(a,r)$ ; $(X$ ; $n(b,r'))$ and in $(n(a,r)$ ; $X)$ ; $n(b,r')$, e.g., $X$ ; $(n(a,r)$ ; $n(b,r'))$.

For a set $E$ of equations such as associativity or associativity-commutativity, the number of elements in an $E$-equivalence class $[t]$ is always finite. But this is not necessarily the case for other equations. For example if $E$ contains the equation $sk(A,pk(A,M)) = M$, then, for the ground term $u = sk(a,pk(a,n(b,r)))$, the number of elements in the equivalence class $[u]$ is infinite: $sk(a,pk(a,n(b,r)))$ and $n(b,r)$ but also any term of the form $sk(a_1,pk(a_1,\ldots,sk(a_n,pk(a_n,sk(a,pk(a,n(b,r)))))\cdots))$. In this case, Maude-NPA keeps only $n(b,r)$, which is called the *normalized* (or *simplified*) version of $u$. Maude-NPA makes a distinction between whether an algebraic property is used for simplification or not, which is explained in detail in Section 3.1.

## 2.2. Equational unification

The execution states associated to a protocol are modeled as elements of an initial algebra $T_{\Sigma/E}$ (i.e., the set of all the terms without variables modulo the algebraic properties $E$). However, Maude-NPA does not work with concrete states in the initial algebra but with $E$-equivalence classes of *symbolic state patterns* $[t(x_1,\ldots,x_n)]$ on the free algebra $T_{\Sigma/E}(X)$ over a set of sorted variables $X$ (i.e., the set of all the terms with variables modulo the algebraic properties $E$).

Maude-NPA relies on equational unification to work with symbolic state patterns. Before we define equational unification, we first need to define the concept of a *substitution*. Given an algebra $T_{\Sigma/E}(X)$, a substitution $\sigma$ on $T_{\Sigma/E}(X)$ is a sort-preserving map from the variables $X$ to $T_{\Sigma/E}(X)$-terms such that $\sigma$ is the identity on all but a finite set of variables. A substitution $\sigma : X \to T_{\Sigma/E}(X)$ extends to a unique $\mathscr{F}$-homomorphism (also denoted by $\sigma$), $\sigma : T_{\Sigma/E}(X) \to T_{\Sigma/E}(X)$ in the obvious way. Given two terms $u$ and $v$ and an equational theory $(\Sigma,E)$ associated to a protocol $\mathscr{P}$, a substitution $\sigma$ is a *E-unifier* of terms $u$ and $v$ (or a unifier modulo $E$) if $\sigma(u) =_E \sigma(v)$. A substitution $\sigma$ is more general than another substitution $\theta$ if $\theta$ is a substitution instance of $\sigma$, i.e., there is a substitution $\gamma$ such that for each variable $x$, $\sigma(x) =_E \gamma(\theta(x))$. *A complete set of most general E-unifiers* of two terms $u$ and $v$ satisfies the property that for any $E$-unifier of $u$ and $v$, there is a unifier in the set more general than it. Unifiers and sets of most general unifiers are defined analogously for systems of equations $\{u_1 = v_1,\ldots,u_k = v_k\}$.

For example, consider that $E$ is just the the cancellation property above, i.e., $sk(A,pk(A,M)) = M$. The complete set of most general $E$-unifiers of the two terms $t = sk(a,X)$ and $s = n(b,r)$ (where $X$ is a variable of sort Msg and $r$ is a variable of sort Fresh) is $\sigma = \{X \mapsto pk(a,n(b,r))\}$.

## 2.3. Maude-NPA states

Maude-NPA performs reachability analysis by: (i) working with symbolic state patterns representing typically *infinite sets* of its ground instances; and (ii) performing equational-unification-based *symbolic execution*. What (i) means is that a pattern $t$ describing a "symbolic state" defines a corresponding *set* $[\![t]\!]$ of actual *instances*. And what (ii) means is that unification works with $E$-equivalence classes of the symbolic states and all the possible $E$-unifiers must be explored.

For example, the Needham-Schroeder public key (NSPK) protocol is specified using the standard Alice-and-Bob notation as follows:

1. $A \to B : pk(B,A;N_A)$
2. $B \to A : pk(A,N_A;N_B)$
3. $A \to B : pk(B,N_B)$

where $A$ and $B$ denote Alice and Bob principal identifiers, $N_A$ and $N_B$ denote the respective nonces, and $pub(A)$ and $pub(B)$ are the respective public keys.

Let us consider an informal representation of the symbolic states found by Maude-NPA, where we include each participant as "*Name:*". The intruder is represented by "*Intruder:*" and contains a *set* of learned messages. Any other participant is labeled by its role label, e.g. "*Alice:*", and contains a *list* of received and sent messages; a positive node $+(m)$ implies sending message $m$, and a negative node $-(m)$ implies receiving $m$.

The following is an informal representation of one of the possible symbolic states found during the execution of the protocol, which represents a partial execution of an instance of Alice's role and an instance of Bob's role:

$$\begin{array}{r l} & \mathit{Sessions}\ \& \\ \text{Alice: } & +(pk(i,A;N_A)), -(pk(A,N_A;N_B)) \\ \text{Bob: } & -(pk(B,A;N_A)), +(pk(A,N_A;N_B)) \\ \text{Intruder: } & \mathit{Knowledge} \end{array}$$

where *Sessions* is a variable denoting uncertain sessions (role executions), *Knowledge* is a variable denoting uncertain intruder knowledge, $A$ and $B$ are variables of sort Name, $i$ is a constant identifying the intruder, and $N_A$ and $N_B$ are variables of sort Nonce.

## 2.4. Reachability analysis

Maude-NPA performs *backwards* symbolic reachability analysis, i.e., if the protocol's usual "forwards" transitions are specified by rules of the form $l \to r$, then the reverse, "backwards" transitions are specified by reversed rules of the form $r \to l$.

One reasonable protocol goal is that, once an honest principal finishes executing an instance of a protocol, apparently with another honest principal, then the intruder should not learn the principal's nonce. This goal, when specified in Maude-NPA, leads to its discovery of the well-known man-in-the-middle attack on Needham-Schroeder. The goal is represented in Maude-NPA by the following attack pattern, where an instance of Bob's role (such instances are referred to as *sessions* )has participated, this session has finished its execution, and the intruder has learned the nonce, $N_B$, generated by this Bob's instance. Logical variables $A_?$ and $N_{A_?}$ are labeled with a question mark, and variables for sessions, or role $\mathit{Sessions}_1$, and the uncertain intruder knowledge, $\mathit{Knowledge}_1$, are written explicitly. All these variables, $A_?$, $N_{A_?}$, $\mathit{Sessions}_1$, and $\mathit{Knowledge}_1$ are existentially quantified, even if the symbol $\exists$ is not written explicitly in the informal attack pattern notation.

$$\begin{array}{r l} & \mathit{Sessions}_1\ \& \\ \text{Bob: } & -(pk(B,A_?;N_{A_?})), +(pk(A_?,N_{A_?};N_B)), -(pk(B,N_B)) \\ \text{Intruder: } & N_B\ \&\ \mathit{Knowledge}_1 \end{array}$$

Then, the *existence of an attack* on the given protocol from a symbolic attack state $u$ like this one exactly means that there is a substitution $\theta$ such that $\theta(u)$ can reach in a backwards direction a state $\theta(v)$ that it is initial (i.e., no more backwards steps can be performed). But this is equivalent to the forwards meaning that from the initial state $\theta(v)$ we can reach $\theta(u)$ in some protocol execution, thus causing an attack on the protocol. The extra ingredients necessary for reachability analysis are just to: (i) incorporate the Dolev-Yao intruder capabilities as transition rules, and (ii) adding new protocol sessions whenever necessary.

Figure 1 shows an informal representation of the full backwards *symbolic* execution from this attack pattern until an initial state is reached. The transition arrows include the honest or dishonest action being performed and the variable instantiation.

Note that although in Figure 1 the attack proceeds from the initial state (at the bottom) to the final state (at the top), the arrows go in the direction of the backwards search. Moreover, as the search proceeds, the variables in the state expressions become further and further instantiated; we attach such instantiations to the arrows. In order to see the attack state that is actually reached, we can compose all the partial substitutions generated through the backwards steps and apply it to the attack pattern, producing the following fully instantiated attack state:

$$\begin{array}{rl} & Sessions_2\ \& \\ \text{Alice:} & +(pk(i,A;N_A)), -(pk(A,N_A;N_B)), +(pk(i,N_B)) \\ \text{Bob:} & -(pk(B,A;N_A)), +(pk(A,N_A;N_B)), -(pk(B,N_B)) \\ \text{Intruder:} & N_B\ \&\ pk(B,N_B)\ \&\ pk(i,N_B)\ \&\ pk(A,N_A;N_B) \\ & \&\ pk(B,A;N_A)\ \&\ pk(i,A;N_A)\ \&\ Knowledge_6 \end{array}$$

It is easy now to figure out the fully instantiated execution path in a forwards sense from the initial state to this fully instantiated attack pattern (just by following Figure 1 from bottom to top).

Let us also illustrate equational unification within this protocol example. In the step called "Intruder extracts $N_B$ from $pk(i,N_B)$", *backwards search* invokes equational unification between the following two terms $N_B$ and $sk(i,M)$, i.e., between the challenge $N_B$ and the intruder action of encrypting a message $M$ using his private key $i$. The equational theory of NSPK is cancellation of encryption and decryption, described by the following equation $sk(A, pk(A,M)) = M$. We find that this unification problem $N_B =^? sk(i,M)$ has a single most general solution $\{M \mapsto pk(i,N_B)\}$ modulo the equational theory of cancellation of encryption and decryption.

## 3. Variant Unification in Maude-NPA

Equational theories found in cryptographic protocols generally contain two types of equations. The first are equations that can be oriented into rules for simplifying terms, e.g. $d(K,e(K,M)) = M$. These rules can be given orientations $t \rightarrow s$ so that $t$ is the more complex term, e.g. $d(K,e(K,M)) \rightarrow M$. These rewrite rules must satisfy certain requirements, which are discussed in more detail in this section. The other equations, called *axioms*, are those that describe theories that are some combination of associative and/or commutative and/or identity axioms. Such axioms have well-known and well-understood unification algorithms. Using the distinction between oriented equations and axioms, the user has a fair amount of freedom for specifying equational theories. In particular, sophisticated users may create their own rewrite rules in order to specify cryptoalgorithms of interest.

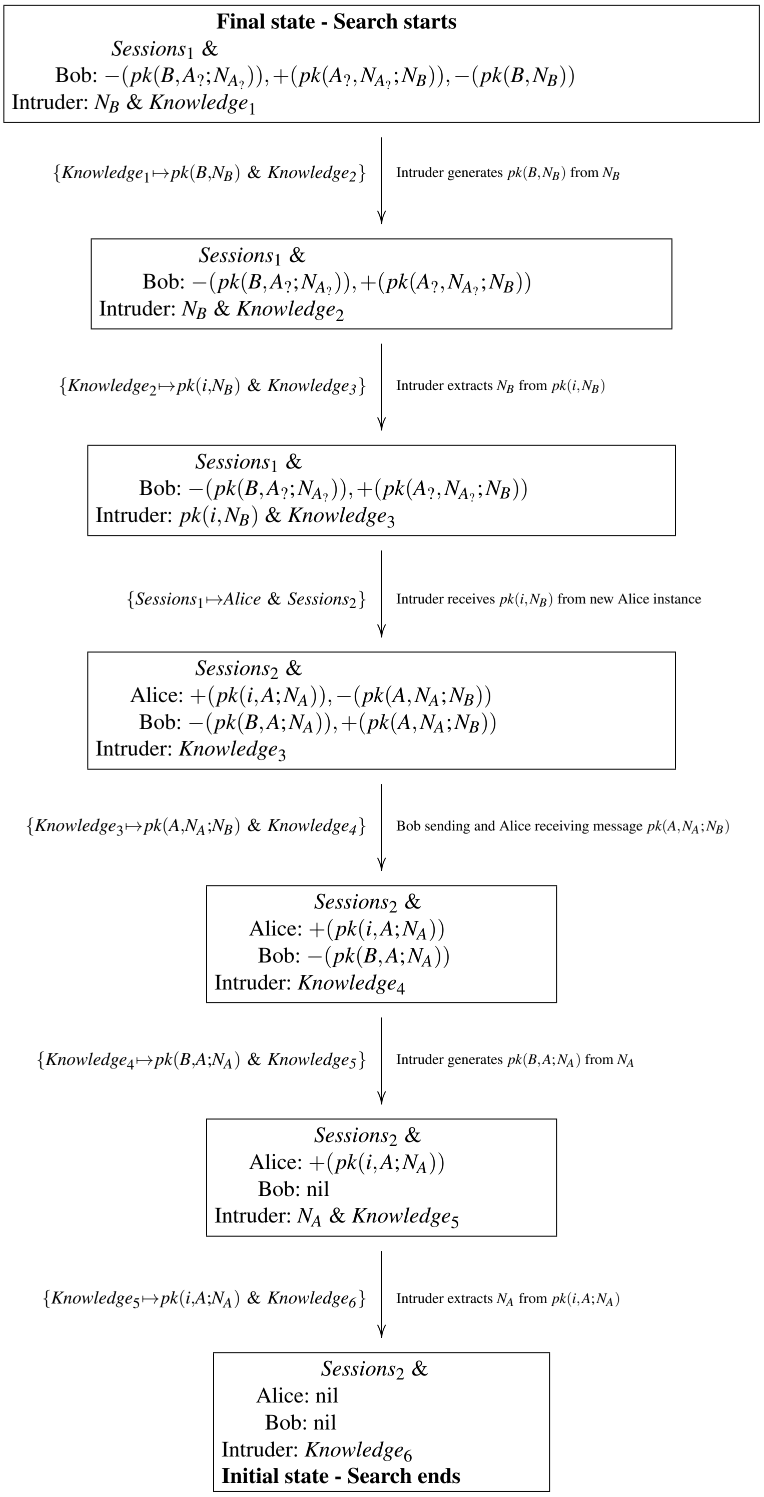

**Final state - Search starts**
$Sessions_1$ &
Bob: $-(pk(B,A_?;N_{A_?})), +(pk(A_?,N_{A_?};N_B)), -(pk(B,N_B))$
Intruder: $N_B$ & $Knowledge_1$
$\{Knowledge_1 \mapsto pk(B,N_B)$ & $Knowledge_2\}$
Intruder generates $pk(B,N_B)$ from $N_B$
$Sessions_1$ &
Bob: $-(pk(B,A_?;N_{A_?})), +(pk(A_?,N_{A_?};N_B))$
Intruder: $N_B$ & $Knowledge_2$
$\{Knowledge_2 \mapsto pk(i,N_B)$ & $Knowledge_3\}$
Intruder extracts $N_B$ from $pk(i,N_B)$
$Sessions_1$ &
Bob: $-(pk(B,A_?;N_{A_?})), +(pk(A_?,N_{A_?};N_B))$
Intruder: $pk(i,N_B)$ & $Knowledge_3$
$\{Sessions_1 \mapsto Alice$ & $Sessions_2\}$
Intruder receives $pk(i,N_B)$ from new Alice instance
$Sessions_2$ &
Alice: $+(pk(i,A;N_A)), -(pk(A,N_A;N_B))$
Bob: $-(pk(B,A;N_A)), +(pk(A,N_A;N_B))$
Intruder: $Knowledge_3$
$\{Knowledge_3 \mapsto pk(A,N_A;N_B)$ & $Knowledge_4\}$
Bob sending and Alice receiving message $pk(A,N_A;N_B)$
$Sessions_2$ &
Alice: $+(pk(i,A;N_A))$
Bob: $-(pk(B,A;N_A))$
Intruder: $Knowledge_4$
$\{Knowledge_4 \mapsto pk(B,A;N_A)$ & $Knowledge_5\}$
Intruder generates $pk(B,A;N_A)$ from $N_A$
$Sessions_2$ &
Alice: $+(pk(i,A;N_A))$
Bob: nil
Intruder: $N_A$ & $Knowledge_5$
$\{Knowledge_5 \mapsto pk(i,A;N_A)$ & $Knowledge_6\}$
Intruder extracts $N_A$ from $pk(i,A;N_A)$
$Sessions_2$ &
Alice: nil
Bob: nil
Intruder: $Knowledge_6$
**Initial state - Search ends**

**Figure 1.** Backwards symbolic execution of man-in-the-middle attack in NSPK

The unification algorithms used by Maude-NPA should satisfy a number of desiderata:

**des1:** They should apply to a large class of theories that occur in cryptographic protocols.
**des2:** If there are any conditions applying to the use of a unification algorithm, they should be easy to verify, preferably by push-button automated means.
**des3:** It should be straightforward to combine unification algorithms for different theories. This is needed because each crypto-algorithm usually has its own theory associated with it, and cryptographic protocols generally use more than one crypto-algorithm.
**des4:** They should support *irreducibility constraints* without losing completeness. . Maude-NPA, as well as other tools, refuses to search any further from states that can be shown to be unreachable. One way of doing this is by looking for a term $u$ in the intruder knowledge that it has proved that the intruder cannot find. These proofs usually require the assumption that $u$ is irreducible, that is, no rewrite rule obtained from $E$ can be applied, even when variables of the term are further instantiated. This assumption is enforced by an irreducibility constraint that requires that any instance $\sigma(u)$ that is reducible by the oriented equations in the equational theory should be rejected.
**des5:** They should be efficient, and the size of the most general set of unifiers produced should be close to the minimum possible. This is because each unifier produces a new backwards analysis step, and thus a new symbolic state. Keeping the number of unifiers small helps to prevent state space explosion.

It is difficult for a class of unification algorithms to satisfy all of these desiderata completely, since several of them are in conflict. For example, there is usually a trade-off between generality (**des1**) and efficiency (**des4**). The algorithm used in Maude-NPA, variant unification, satisfies (**des1**) through (**des4**). It is less successful with (**des5**), but it is still efficient enough to be practical in a large number of cases. We will discuss possible methods for improving efficiency, in particular reducing the size of the most general set of unifiers found, in Section 6.

## 3.1. Terms, Positions, Rewriting and Narrowing

Here we give some basic definitions involving rewriting and narrowing [3,4].

In order to define rewriting and narrowing, we will first need to find positions in a term. *Positions* are strings of natural numbers used to locate subterms of a term, and are defined recursively. If $t$ is a term, then the position of $t$ in $t$ is denoted by the empty string $\varepsilon$. If $s$ is located at position $p$ in $t$, and $u$ is the $i$'th argument of $s$, then $u$ is located at position $p.i$. We use $t|_p$ to denote the subterm of $t$ located at $p$, and $s = t[u]_p$ to denote the result of replacing the subterm of $t$ at positon $p$ with $u$. We say that $p$ is a *variable position* of $t$ if $t|_p$ is a variable. Thus, if $t = f(g(x,y),z)$, then $t|_\varepsilon = f(g(x,y),z)$, $t|_1 = g(x,y)$, $t|_2 = z$, $t|_{1.1} = x$, and $t|_{1.2} = y$.

Given an equational theory $(\Sigma, R)$ consisting of rewrite rules, and a term $t$, we say that $t$ *rewrites* to $s$ via the rewrite rule $\ell \to r$ if there is a position $p$ of $t$ such that there is a matching substitution $\theta$ of $t|_p$ and $\ell$ (i.e., $\theta(l) = t|_p$), and $s = t[\theta(r)]_p$. We denote this by $t \overset{\sigma,p}{\to}_{\ell\to r} s$ (or simply $t \to_{\ell\to r} s$). We say that $t_1$ rewrites to $t_k$ with a rule in $R$, $t_1 \to_R^* t_k$, if there is a sequence $t_1,\ldots,t_i,\ldots,t_k$ such that $t_i \to_{\ell_i \to r_i} t_{i+1}$ such that $\ell_i \to r_i \in R$. We say that $t$ is *irreducible* if there is no $s$ such that $t \to_R s$.

If a term $t$ has variables, we can also apply *narrowing* with the rules $R$ to it. We say that $t$ *narrows* to $s$ via the rewrite rule $\ell \to r$ if there is a non-variable position $p$ of $t$ and a unifier $\sigma$ of $t|_p$ and $\ell$ (i.e., $\sigma(t|_p)$ and $\sigma(\ell)$ are equal), and $s = \theta(t[r]_p)$. We denote this by $t \overset{\sigma,p}{\rightsquigarrow}_{\ell\to r} s$ (or simply $t \rightsquigarrow_{\ell\to r} s$). We say that $t_1$ narrows to $t_k$ via $R$, $t_1 \overset{\sigma}{\rightsquigarrow}{}^{*}_{R} t_k$, if there is a sequence $t_1,\ldots,t_i,\ldots,t_k$ such that $t_i \overset{\sigma_i}{\rightsquigarrow}_{\ell_i\to r_i} t_{i+1}$, $\ell_i \to r_i \in R$, and $\sigma = \sigma_1 \cdots \sigma_{k-1}$.

We will be particularly interested in rewriting and narrowing *modulo axioms*. This is a form of rewriting and narrowing in which the equational theory is of the form $(\Sigma, E \uplus B)$, the disjoint union of $E$ and $B$, where $E$ is a set of rewrite rules and $B$ is a set of axioms. We write $t \to_{E/B} s$ if there are terms $t'$ and $s'$ such that $t =_B t'$, $t' \to_E s'$, and $s =_B s'$. Suppose, for example, that a $+$ symbol has been declared commutative with the `comm` attribute, and that we have an equation in $E$ of the form $x+0 = x$. Then we can apply such an equation to the term $0+7$ *modulo* commutativity, even though the constant 0 is on the left of the $+$ symbol. That is, the term $0+7$ *matches* the left-hand side pattern $x+0$ *modulo* commutativity. We would express this rewrite step of simplification modulo commutativity with the arrow notation:

$$0+7 \to_{E/B} 7$$

where $E$ is the set of equations containing the above equation $x+0 = x$, and where $B$ is the set of axioms containing the commutativity of $+$. Likewise, we denote by $\to^*_{E/B}$ the reflexive-transitive closure of the one-step rewrite relation $\to_{E/B}$ with the equations $E$ modulo the axioms $B$. That is, $\to^*_{E/B}$ corresponds to taking zero, one, or more rewrite steps with the equations $E$ modulo $B$.

The rewriting relation $\to_{E/B}$ is difficult to implement (it is in fact generally undecidable), and so we prefer to use a simpler form of rewriting performed on representatives of $B$-equivalence classes of subterms. We say that $t \to_{E,B} s$ if there is a position $p$ of $t$ such that there is a $B$-matching substitution $\theta$ of $t|_p$ and $\ell$ (i.e., $t|_p = \theta(\ell)$), and $s = t[\theta(r)]_p$. Soundness and completeness of $E,B$-rewriting with respect to $E/B$-rewriting requires a condition called *strict coherence* [8]. Consider, for example, an exclusive or operator $\oplus$ which has been declared *AC*. Now consider the equation $x \oplus x = 0$. This equation, if not completed by another equation, is *not* coherent modulo AC. What this means is that there will be term *contexts* in which the equation *should* be applied, but it *cannot* be applied. Consider, for example, the term $b \oplus (a \oplus b)$. Intuitively, we should be able to apply the above equation to simplify the term above to the term $a \oplus 0$ in one step. However, we cannot! The problem is that the equation cannot be applied (even if we match modulo *AC*) to either the top term $b \oplus (a \oplus b)$ or the subterm $a \oplus b$. We can however make our equation coherent modulo *AC* by adding the extra equation $x \oplus x \oplus y = 0 \oplus y$, which we can slightly simplify to the equation $x \oplus x \oplus y = y$ by using the equation $x \oplus 0 = x$. This extended version of our original equation will now apply to the term $b \oplus (a \oplus b)$, giving the simplification $b \oplus (a \oplus b) \longrightarrow_{E,B} a$. The methods for guaranteeing strict coherence for the axioms sets $B$ of interest to us are described in detail in Section 3.2.1.

Finally, $(E,B)$-narrowing is defined similarly to $(E,B)$-rewriting but using $B$-unification of $t|_p$ and $\ell$ instead of $B$-matching.

### 3.2. Requirements for Variant Algebraic Theories

As we will show, for theories $(\Sigma, E)$ which can be decomposed into a set $B$ of axioms and a set $E$ of oriented equations and satisfy the requirements explained in this section, Maude-NPA uses a technique called *folding variant narrowing* [6] to perform unification of symbolic terms *modulo* the oriented equations $E$ and the axioms $B$ specified for the algebraic properties of the protocol. In order for this folding variant narrowing strategy to be sound and complete and to provide a *finite* set of unifiers, six specific requirements must be met by any algebraic theory specifying cryptographic functions that the user provides. If these requirements are not satisfied, Maude-NPA may exhibit non-terminating and/or incomplete behavior, and any completeness claims about the results of the analysis can no longer be guaranteed. We list below these six requirements and explain in detail what they mean. We consider *algebraic theories* $T$ in which axioms and rules are explicitly *decomposed* in the form: $T = (\Sigma, B, E)$, where $\Sigma$ is a *signature* declaring sorts, subsorts, and function symbols, and $E \uplus B$ is the disjoint union of a set $B$ of equational axioms such as our previous combinations of associativity and/or commutativity and/or identity axioms, and a set $E$ of oriented equations to be used from left to right as rewrite rules.

In Maude-NPA we call an algebraic theory $T = (\Sigma, B, E)$ specified by the user for the cryptographic functions of the protocol *admissible* if it satisfies the following six requirements:

1. The axioms $B$ can declare some binary operators in $\Sigma$ to have any combination of associativity (A), commutativity (C), and identity (U) axioms.
2. The equations $E$ are confluent modulo $B$.
3. The equations $E$ are terminating modulo $B$.
4. The equations $E$ are coherent modulo $B$ (see [5]).
5. The equations $E$ are sort-decreasing.
6. The equations $E$ have the finite variant property (see [6,7]).

We now explain in detail what these requirements mean.

#### 3.2.1. Guaranteeing Strict Coherence

Here, we show how to guarantee strict coherence for the three theories most commonly used in Maude-NPA: $AC$, $ACU$, and $A$:

1. For any symbol $f$ which is $AC$, and for any equation of the form $f(u,v) = w$ in $E$, we just add the equation $f(f(u,v),x) = f(w,x)$, where $x$ is a fresh new variable.
2. If $f$ is $ACU$ with identity symbol $e$, the original equation $f(u,v) = w$ is *replaced* by the extended equation $f(f(u,v),x) = f(w,x)$ shown above, instead of being added.
3. Likewise, if $f$ is associative only, the following *extended equations* should be added: $f(x, f(u,v)) = f(x,w)$, $f(f(u,v),y) = f(w,y)$, and $f(x, f(f(u,v),y)) = f(x, f(w,y))$.

In an order-sorted setting, we should give to such new variables $x$ and $y$ *the biggest sort possible*, so that they will apply in all generality.

As an additional optimization, note that some equations may already be coherent modulo $B$, so that we need not add the extra equation or equations. This can be checked by determining if the new equation $s = t$ can be derived from the already existing equations. Consider for example the exclusive-or theory $x \oplus 0 = x$, $x \oplus x = 0$ and $x \oplus x \oplus y = y$, where $\oplus$ is AC. Consider the extended equation $(x \oplus 0) \oplus z = x \oplus z$ constructed using $x \oplus 0$ and 1) from above to guarantee coherence. Since $x \oplus 0 = x$, we have $(x \oplus 0) \oplus z = x \oplus z$, so the equation already follows from the original theory.

Also, if we assume that symbol $\oplus$ is ACU instead of AC, then the previous three equations $x \oplus 0 = x$, $x \oplus x = 0$ and $x \oplus x \oplus y = y$ for $\oplus$ being AC can be simplified into one equation $x \oplus x \oplus y = y$ for $\oplus$ being ACU with 0 as the identity symbol. Note that this equation is coherent modulo *ACU* but it is not terminating modulo *ACU* since for any term $T$, we have $T =_{ACU} 0 \oplus 0 \oplus T$ and $0 \oplus 0 \oplus T \rightarrow_{E/ACU} T$ using the equation $x \oplus x \oplus y = y$. Because of this, exclusive-or must be specified in Maude-NPA with an AC symbol and never with an ACU symbol.

### 3.2.2. *Sort-decreasingness*

An equation $u = v$ is *sort-decreasing* iff for any substitution instance $u\theta = v\theta$, if $v\theta$ is of sort s, then $u\theta$ must also be of sort s. This helps us avoid instances in which a term containing $u$ may obey sort restrictions, but not when $u$ is replaced by $v$.

As an example of what can happen when a theory is not sort-decreasing, suppose that for the cancellation of encryption and decryption we use a subsort Encoding of the sort Msg and the following definitions for symbols *pk* and *sk*:

```
op pk : Name Msg -> Encoding [frozen] .
op sk : Name Msg -> Encoding [frozen] .
```

Then the following equations for cancellation are *not* sort-decreasing, since the left-hand sides are defined as elements of sort Encoding but the application of the equations may return elements of a greater sort Msg:

```
eq pk(A:Name,sk(A:Name,Z:Msg)) = Z:Msg [variant] .
```

Suppose that we have defined a function symbol $f$ : Encoding $\rightarrow$ Msg. Now, the term $f(pk(a,sk(a,n(a,r))))$ appears to obey sort restrictions, since $pk(a,sk(a,n(a,r)))$ is of sort Encoding. But $pk(a,sk(a,n(a,r)))$ is equivalent to $n(a,r)$, which is of sort Nonce, and $f(n(a,r))$ is not well-typed. The problem could have been avoided if *pk* had been declared of sort Msg.

### 3.2.3. *Confluence*

The equations $E$ are called *confluent* modulo $B$ if and only if for each term $t$ in the theory $T = (\Sigma, B, E)$, if we can rewrite $t$ with $E$ modulo $B$ in two different ways to terms $u$ and $v$, then we can always further rewrite $u$ and $v$ to a common term modulo $B$. That is, when there are terms $u$ and $v$ such that $t \longrightarrow^{*}_{E,B} u$ and $t \longrightarrow^{*}_{E,B} v$, we can always find terms $u', v'$ such that $u \longrightarrow^{*}_{E,B} u'$ and $v \longrightarrow^{*}_{E,B} v'$, and $u' =_B v'$. Note that $u'$ and $v'$ are thought of as the same term, in the sense that they are equal modulo the axioms $B$. In our above example we have, for instance, $0 + 7 =_B 7 + 0$.

### 3.2.4. Termination

The equations $E$ are called *terminating* modulo $B$ if and only if all rewrite sequences terminate; that is, if and only if we never have an infinite sequence of rewrites

$$t_0 \rightarrow_{E,B} t_1 \rightarrow_{E,B} t_2 \ldots t_n \rightarrow_{E,B} t_{n+1} \cdots$$

### 3.2.5. The Finite Variant Property

In this section we define variants and the finite variant property, first introduced by Comon and Delaune in [9]. The finite variant property is essential to Maude-NPA 3.x's variant unification algorithm.

Recall that, given a theory $(\Sigma, E \uplus B)$, where $E$ is a set of rewrite rules and $B$ is a set of axioms, we say that a term $t$ in $\mathscr{T}_\Sigma(X)$ is *normalized* or *irreducible* if no rewrite rules from $E$ can be applied to any members of the $B$-equivalence class $t$ belongs to. We say that such a normalized $t$ is a *normal form* of $s$, if $t$ can be produced from $s$ via a finite number of $E,B$ rewriting steps. Furthermore, if $(\Sigma, B, E)$ satisfies the first five admissibility conditions given in Section 3.2, then every term in $\mathscr{T}_\Sigma(X)$ has a unique normal form modulo $B$, which can be found after a finite number of decidable rewriting steps. We refer to this (unique by confluence) normal form of t as $t{\downarrow_{E,B}}$. We can then define the set of $(E,B)$-*variants* of a term $t$ as the set of all pairs of the form $(\sigma, \sigma(t){\downarrow_{E,B}})$ where $\sigma$ is a substitution and $\sigma(t){\downarrow_{E,B}}$ is the normal form of $\sigma(t)$.

For example, given the equational theory $(\Sigma, E \uplus B)$ for exclusive-or shown in Section 3.2.1, and the term $X{:}Msg \oplus Y{:}Msg$, we can construct several of its variants as follows:

1. The pair $(\{X{:}Msg \mapsto a \oplus b, Y{:}Msg \mapsto a \oplus b,\ a \oplus b \oplus a \oplus b)$ is normalized to $(\{X{:}Msg \mapsto a \oplus b, Y{:}Msg \mapsto a \oplus b, 0)$;
2. The pair $(\{X{:}Msg \mapsto a \oplus b \oplus U{:}Msg, Y{:}Msg \mapsto a \oplus b, a \oplus b \oplus U \oplus a \oplus b)$ is normalized to $(\{X{:}Msg \mapsto a \oplus b, Y{:}Msg \mapsto a \oplus b, U)$, and;
3. The pair $(\{X{:}Msg \mapsto a \oplus b \oplus U{:}Msg, Y{:}Msg \mapsto a \oplus b \oplus V, a \oplus b \oplus U{:}Msg \oplus a \oplus b \oplus V{:}Msg)$ is normalized $(\{X{:}Msg \mapsto a \oplus b, Y{:}Msg \mapsto a \oplus b\}, U{:}Msg \oplus V{:}Msh)$.

We say that a variant $(\theta_1, t_1)$ of $t$ is *more general* than another variant $(\theta_2, t_2)$ of $t$ if there is a substitution $\rho$ such that

1. $\rho(t_1) =_B t_2$, and;
2. $\theta_2 =_B (\theta_1 \rho){\downarrow_{E,B}}$.

Thus, the variant $(\{X \mapsto Z, Y \mapsto Z\}, Z \oplus Z{\downarrow_{E,B}})$ is strictly more general than $(\{X \mapsto a \oplus b, Y \mapsto a \oplus b\}, (a \oplus b \oplus a \oplus b){\downarrow_{E,B}})$ even though both $Z \oplus Z$ and $a \oplus b \oplus a \oplus b$ normalize to the same term 0, because $\{X \mapsto Z, Y \mapsto Z\}$ is strictly more general than $\{X \mapsto a \oplus b, Y \mapsto a \oplus b\}$.

A set of variants $V_t$ of a term $t$ with the property that for every variant $(\sigma, t\sigma)$ of $t$, there is a more general variant of $t$ in $V_t$, is called a *set of most general* variants of $t$. For example, the following is a set of most general variants of $a \oplus V$, where $a$ is a constant and $V$ is a variable:

$$\{(id, a \oplus V), (\{V \mapsto a \oplus U\}, U), (\{V \mapsto 0\}, a), (\{V \mapsto a\}, 0\}$$

Given a theory $T = (\Sigma, E \uplus B)$, we say the decomposition $(\Sigma, B, E)$ has the *finite variant property* if for every term $t$, there is a finite set $V_t$ of most general $(E,B)$-variants of $t$. Let us illustrate this concept with a positive and a negative example of theories having, or failing to have, the finite variant property.

The equational theory for exclusive-or shown in Section 3.2.1 does have the finite variant property. For example, the term $X \oplus Y$ has a set of seven most general variants as produced using the Maude command "`get variants`" command. These are shown in Table 1.

| Substitution $\sigma$ | | Normalized Instance $\sigma(X \oplus Y)$ |
|---|---|---|
| $X \mapsto U$ | $Y \mapsto V$ | $U \oplus V$ |
| $X \mapsto Z \oplus U$ | $Y \mapsto Z \oplus V$ | $U \oplus V$ |
| $X \mapsto U \oplus V$ | $Y \mapsto V$ | $U$ |
| $X \mapsto V$ | $Y \mapsto U \oplus V$ | $U$ |
| $X \mapsto U$ | $Y \mapsto U$ | $0$ |
| $X \mapsto 0$ | $Y \mapsto U$ | $U$ |
| $X \mapsto U$ | $Y \mapsto 0$ | $U$ |

**Table 1.** Variants of $X \oplus Y$ produced by Maude "`get variants`" command

The key idea for the finite variant property is that, given a term $t$ and a (normalized) substitution $\theta$, any pair $(\theta, \theta(t){\downarrow}_{E,B})$ must be either equal (up to renaming) to or less general than, i.e., a further instantiation of, some variant of $t$. For example, the substitution $X \mapsto a \oplus b \oplus U, Y \mapsto a \oplus b \oplus V$ maps $X \oplus Y$ to a term that reduces to $U \oplus V$. This variant is a special case of the second variant of Table 1 produced by Maude-NPA. If we then consider a further instantiation $\{\mathtt{U} \mapsto \mathtt{c} \ \oplus \ \mathtt{d}, \mathtt{V} \mapsto \mathtt{c} \ \oplus \ \mathtt{d}\}$ of the term $\mathtt{a} \oplus \mathtt{b} \oplus U \oplus \mathtt{a} \oplus \mathtt{b} \oplus V$, then the term $\mathtt{a} \oplus \mathtt{b} \oplus \mathtt{c} \oplus \mathtt{d} \oplus \mathtt{a} \oplus \mathtt{b} \oplus \mathtt{c} \oplus \mathtt{d}$ is simplified into 0, with the composed substitution $X \mapsto a \oplus b \oplus c \oplus d, Y \mapsto a \oplus b \oplus c \oplus d$. This is a special case of the fifth variant of Table 1 produced by Maude-NPA.

The question of whether or not an equational theory has the finite variant property is undecidable [17]. However, if the theory *does* have the finite variant property, it is quite easy to check that it does; and if it *does not*, it is also quite easy to get strong empirical evidence suggesting that it either it does not, or the number of variants is so large that they are not practical to compute.

First of all, there is a semi-decision procedure, the *folding variant narrowing strategy* of [6] for producing all the variants of a single term $t$. The idea, is that, if $(\sigma, t')$ is a variant of $t$, and $t' \overset{\rho}{\rightsquigarrow}_{R,E} t''$, then $(\sigma\rho, t'')$ is also a variant of $t$. Moreover, any variant of $t$ is a special case of some $(\theta, t'')$ produced by iterative narrowing. In folding variant narrowing, we produce a narrowing tree, but do not add any new leaves that are equal to (up to change in variables) or special cases of nodes already in the tree. The resulting tree gives a complete set of variants of $t$, and is finite if and only if any term $t$ has a finite number of variants.

A semi-decision procedure for checking the finite variant property that works well in practice together with folding variant narrowing was introduced in [7]. The procedure is as follows: for each function symbol $f$ in $\Sigma$, compute the variants of $f(X_1, \ldots, X_n)$, where $n$ is the arity of $f$ and the $X_i$ are variables. A theory has the finite variant property, if and only if a finite number of variants is returned for each such function symbol, that is, the set of variants obtained by the `get variants` command for the term $f(X_1 : A_1, \ldots, X_n :$

$A_n$) is finite. We note that, if the theory does not have the finite variant property, Maude will simply keep returning more and more variants. The user may then conclude that either the theory does not have the finite variant property, or it may do so but it produces too many variants to be practical.

### *3.2.6. Variant Unification*

We are now ready to define variant unification for a theory decomposition $(\Sigma, B, E)$ having the finite variant property. Assume that the $n$ connected components of the poset of sorts $(\mathsf{S}, <)$ considered as a directed graph have each a top sort, say, $\mathsf{s}_1, \ldots, \mathsf{s}_n$. This can be safely assumed because if no top sort exists in a connected component, it can be added, as done automatically by Maude. Then define a new fresh sort Truth with a constant $\mathtt{tt} : \to \mathsf{Truth}$ not related to any other sort and $n$ "equality predicates" $\mathtt{eq} : \mathsf{s_i}\ \mathsf{s_i} \to \mathsf{Truth}$, $1 \leq i \leq n$. Furthermore, add $n$ equations $\mathtt{eq}(x_i, x_i) = \mathtt{tt}$, with $x_i$ of sort $\mathsf{s_i}$, $1 \leq i \leq n$. This extension gives us a new equational theory decomposition $(\Sigma^{\mathtt{eq}}, B, E^{\mathtt{eq}})$, where $E^{\mathtt{eq}}$ is obtained by adding the above equality predicate equations to $E$. The theory $(\Sigma^{\mathtt{eq}}, B, E^{\mathtt{eq}})$ also has the finite variant property, since it satisfies the same executability requirements as $(\Sigma, B, E)$, the variants of the function symbols in $\Sigma$ remain the same, and each term $\mathtt{eq}(x_i, x_i')$ has two variants, namely, $(id, \mathtt{eq}(x_i, x_i'))$ and $(\{x_i \mapsto x_i'', x_i' \mapsto x_i''\}, \mathtt{tt})$. Now note that given an equation $u = v$, and a substitution $\theta$ by termination and confluence of the rewriting relation and the definition of the equality predicates, we have:

$$u\theta =_{E \cup B} v\theta \Leftrightarrow (u\theta)\downarrow_{E,B} =_B (v\theta)\downarrow_{E,B} \Leftrightarrow \mathtt{eq}(u\theta, v\theta)\downarrow_{E,B} = \mathtt{tt}$$

Therefore, the subset of variants of $\mathtt{eq}(u, v)$ of the form $(\theta, \mathtt{tt})$ provides a complete set of most general unifiers of the equation $u = v$.

For example, consider the problem $X \oplus a = Y \oplus b$. We can add the eq symbol and the equation $\mathtt{eq}(x, x) = \mathtt{tt}$ to the exclusive-or shown in Section 3.2.1 and generate the variants of the term $\mathtt{eq}(X \oplus a, Y \oplus b)$ using the "`get variants`" command of Maude [14], which returns 23 variants but we show only those returning `tt`.

```
Maude> get variants eq(X * a, Y * b) .

Variant #2           Variant #3         Variant #10       Variant #13
Bool: tt             Bool: tt           Bool: tt          Bool: tt
X --> b * %1         X --> b            X --> a           X --> a * b * #1
Y --> a * %1         Y --> a            Y --> b           Y --> #1

Variant #20          Variant #21        Variant #22       Variant #23
Bool: tt             Bool: tt           Bool: tt          Bool: tt
X --> #1             X --> a * %1       X --> a * b       X --> 0
Y --> a * b * #1     Y --> b * %1       Y --> 0           Y --> a * b
```

This is exactly the same as the "`variant unify`" command in Maude [14].

```
Maude> variant unify X * a =? Y * b .

Unifier #1           Unifier #2         Unifier #3       Unifier #4
X --> b * %1         X --> b            X --> a           X --> a * b * #1
Y --> a * %1         Y --> a            Y --> b           Y --> #1
```

```
Unifier #5          Unifier #6       Unifier #7      Unifier #8
X --> #1             X --> a * %1      X --> a * b      X --> 0
Y --> a * b * #1     Y --> b * %1      Y --> 0          Y --> a * b
```

## 4. A Case Study: The YubiKey and YubiHSM APIs

In this section, we present a case study of a complex cryptographic API [1] described in Maude-NPA that involves a number of different features, all handled by variant unification.

Yubico is a leading company on open authentication standards and has developed two core inventions: the *YubiKey*, a small USB designed to authenticate a user against network- based services, and the *YubiHSM*, Yubico's hardware security module (HSM). The YubiKey allows for the secure authentication of a user against network-based services by considering different methods: one-time password (OTP), public key encryption, public key authentication, and the Universal 2nd Factor (U2F) protocol. YubiKey works by using a secret value (i.e., a running counter) and some random values, all encrypted using a 128 bit Advanced Encryption Standard (AES). An important feature of YubiKey is that it is independent of the operating system and does not require any installation, because it works with the USB system drivers. YubiHSM is intended to operate in conjunction with a host application. It supports several modes of operation, but the key concept is a symmetric scheme where one device at one location can generate a secure data element in a secure environment. Although the main application area is for securing YubiKey's OTP authentication/validation operations, the use of several generic cryptographic primitives allows a wider range of applications. The increasing success of YubiKey and YubiHSM has led to its use by governments, universities and companies like Google, Facebook, Dropbox, CERN, Bank of America etc., including more than 30,000 customers.

Cryptographic Application Programmer Interfaces (Crypto APIs) are commonly used to secure interaction between applications and hardware security module (HSMs), and are used in both YubiKey and YubiHSM. However, many crypto APIs have been subjected to intruder manipulation to disclose relevant information, as is the case for YubiHSM. In [11], Künnemann and Steel show two kinds of attacks on the first released version of the YubiHSM API.

In [1], we were able to both prove security properties of YubiKey generation 2 and find the two attacks on version 0.9.8 of YubiHSM in a completely automated way beyond the analysis reported in [11]. Note that there has not been any completely automated analysis of these two attacks before, because both YubiKey and YubiHSM involve a number of complex challenges:

(1) handling of Lamport clocks,
(2) modeling of mutable memory,
(3) handling of constraints on the ordering of events, and
(4) support for symbolic reasoning modulo exclusive or.

We performed the analysis of these APIs in a fully-unbounded session model making no abstraction or approximation of fresh values, and with no extra assumption, or user interaction.

The main goal of [1] was to investigate whether Maude-NPA could complement and extend the formal modeling and analysis results about YubiKey and YubiHSM obtained in [11,12]. This is a non-obvious question: on the one hand, Maude-NPA has provided support for exclusive-or for years, so it is well-suited for meeting Challenge (4). But, on the other hand, previous applications of Maude-NPA have not addressed Challenges (1)-(3). The main upshot of the results we presented can be summarized as follows: Challenge (2) can be met by expressing mutable memory in terms of synchronization messages, a notion used in Maude-NPA to specify protocol compositions [13]; Challenge (3) can be met by the recently added unification modulo associativity [14,15], allowing an easy treatment of lists; and Challenge (1) can be met by a slight extension of Maude-NPA's current support for equality and disequality constraints [16], namely, by adding also support for constraints in Presburger Arithmetic. In this way, we show how challenges (1)-(4) can all be met by Maude-NPA, and how these results in automated formal analyses of YubiKey and YubiHSM substantially extend previous analyses. Very few tools are well equipped to simultaneously handle all these challenges.

It is important to remark that all four Challenges are handled by variant unification: Challenge 1 is met by adding Presburger Arithmetic constraints described by specifying $+$ as an *ACU*-symbol in Maude-NPA, Challenge 2 is met by providing *ACU*-symbols for defining the mutable memory, Challenge 3 is met by providing *A*-symbols for defining lists of events, and Challenge 4 is met by using the exclusive-or specification of Section 3.2.1. Even more, variant unification in Maude-NPA *must* take care of all four at the same time. Reasoning about such a complex theory is not possible for any other protocol analysis tool we are aware of.

## *4.1. The YubiKey and YubiHSM authentication devices*

The YubiKey USB device is an authentication device capable of generating One Time Passwords (OTPs). The YubiKey connects to a USB port and identifies itself as a standard USB device such as a keyboard, which allows it to be used in most computing environments using the system's native drivers. In the YubiKey OTP mode, there is a button physically located on the YubiKey and, when this button is pressed, it emits a string that can be verified only once against a server in order to receive the permission to access a service. Furthermore, a request for a new authentication token is triggered also by touching the YubiKey button. As a result of this request, some counters that are stored on the device are incremented and some random values are generated in order to create a fresh 16-byte plaintext.

Each OTP sent by the YubiKey is encrypted using an AES key. Thus, the YubiKey authentication server accepts an OTP only if both it decrypts under the appropriate AES key and the token counter stored in the OTP is larger than the token counter stored in the last OTP received by the server. The token counter is used as a Lamport clock, i.e., it is used to determine the order of events in a distributed concurrent system by using a counter that both has a minimum value (e.g., 0) and has a minimum tick (increment of the counter).

Yubico also distributes a USB device that works as an application-specific Hardware Security Module (HSM) to protect the YubiKey AES keys. The YubiHSM [10] stores a very limited number of AES keys so that the server can use them to perform cryptographic operations without the key values ever appearing in the server's memory. The

YubiHSM is designed to protect the YubiKey AES keys when an authentication server is compromised by encrypting the AES keys using a master key stored inside the YubiHSM.

The AES keys are only readable to the YubiHSM through the use of *Authenticated Encryption with Associated Data (AEAD)*. The AEAD uses a cryptographic method that provides both confidentiality and authenticity. An AEAD consists of two parts: (i) the encryption of a message using the counter mode cryptographic mode of operation, and (ii) a *message authentication code (MAC)* taken over the encrypted message. In order to construct, decrypt or verify an AEAD, a symmetrical cryptographic key and a piece of associated data are required. This associated data, called a *nonce* in the rest of the paper, can either be a uniquely generated handle or something that is uniquely related to the AEAD. Unlike the more commonly used definition of nonce, a nonce is not required to be unpredictable, so an intruder could be able to reconstruct it.

To encrypt a message using counter mode, one first divides it into blocks of equal length, each suitable for input to the block cipher AES, e.g. $data_1, \ldots, data_n$. The sequence $counter_1, \ldots, counter_n$ is then computed, where $counter_i = nonce \oplus i$ modulo $2^\eta$ and $\eta$ is the length of a block in bits. The encrypted message is then $senc(counter_1, k) \oplus data_1; \ldots; senc(counter_n, k) \oplus data_n$, where $senc$ is the encryption function and $k$ the symmetrical cryptographic key, and $senc(counter_1, k); \ldots; senc(counter_n, k)$ is called the *keystream*. Finally, the MAC is computed over the encrypted message and appended to obtain $(senc(counter_1, k) \oplus data_1; \ldots; senc(counter_n, k) \oplus data_n); MAC$. The MAC is of fixed length, so it is possible to predict where it starts in an AEAD. However, we follow the generalization of [12] and consider just AEADs of the form $senc(cmode(nonce), k) \oplus data; mac(data, k)$.

In [11,12], Künnemann and Steel reported two kinds of attacks on version 0.9.8 beta of YubiHSM API: (a) if the intruder has access to the server running YubiKey, where AES keys are generated, then it is able to obtain plaintext in the clear; (b) even if the intruder has no access to the server running YubiKey, it can use previous nonces to obtain AES keys.

Attack (b) is the more interesting of the two, so we present it here. It involves a YubiHSM command, AEAD-Generate, that takes as input a nonce, a handle to an AES key, some data and outputs a *AEAD*. An intruder can produce an *AEAD* for the same handle *kh* and a value *nonce* that was previously used to generate another *AEAD*. An intruder can recover the keystream directly by using the AEAD-Generate command to encrypt a string of zeros and discarding the *MAC*. The result will be the exclusive-or of the keystream with a string of zeros, which is equal to the keystream itself.

So, the steps followed are

1. Intruder observes AEAD $senc(cmode(nonce), k) \oplus data; mac(data, k)$ and removes $mac(data, k)$.
2. Intruder uses same nonce and handle to have YubiHSM provide an AEAD for a string of zeros: $senc(cmode(nonce), k) \oplus 0; mac(0, k)$ and removes $mac(data, k)$.
3. Intruder computes

$$(senc(cmode(nonce), k) \oplus data) \oplus (senc(cmode(nonce), k) \oplus 0) = data \oplus 0 = data$$

We note that this attack depends both on the equational properties of exclusive-or, and on the mutable global memory that is accessed by YubiHSM when it retrieves the key $k$ corresponding to the handle. In [1], Maude-NPA was able to find both attacks automatically. This goes beyond the analysis reported in [11], which needed auxiliary user-defined lemmas and had limited support for exclusive-or.

No attacks were found on the Yubikey module , but many of the properties that were verified involve a combination of global mutable memory, conditions on sequences of events, and the use of Lamport clocks. For example, one of the properties specifies absence of replay attacks, which is formulated as the conditions that no two logins accept the same counter value. This involves conditions on sequences of events (no two different logins accepting the same counter value should appear in a sequence), global mutable memory (to store the counter values) and Lamport clocks (to model the means by which counter values are updated). In [11] three such security properties of Yubikey were specified and proven; in [1] they were verified using Maude-NPA.

We now describe how the different challenges are handled in Maude-NPA, and we describe the different equational theories they introduce.

## 4.2. Modeling global memory

Following [11], all predicates are stored together in a shared global memory. Some predicates are read-only, but others are updated. Maude-NPA, unlike some other tools, does not natively support mutable memory; but it can be modeled using a multiset of predicates allocated in input and output strand synchronization messages (see [13]). That is, the old data will appear in the input synchronization message of an API strand, and the new information will appear in the output synchronization message of that strand, which will then become the input synchronization message of the next API strand. The multiset of predicates is defined using a new symbol `@`, which is an infix associative-commutative symbol with an identity symbol `empty`. Thus, for the strand describing the YubiKey button press, the input synchronization message is as follows:

```
{yubikey -> yubikey ;; 1-1 ;;
 Y(pid,sid) @ YubiCounter(pid,c1) @ Server(pid,sid,c2) @ SharedKey(pid,k)}
```

Updating the counter of the YubiKey after a button press is represented by updating the second argument of the `YubiCounter(pid,c1)` predicate in the multiset. This updated multiset becomes the output synchronization of the strand.

## 4.3. Modeling lists of events

The YubiKey API also keeps a rigid control of the ordering of events, where an event is a state transition in the system, and a proper analysis of actions is mandatory. Maude-NPA, unlike other tools, does not natively support the representation and analysis of event sequences; but we have implemented it by storing event sequences in the synchronization messages. This is helped by the fact that Maude-NPA, via the Maude language, has recently been endowed with unification modulo associativity [14,15]. In particular, this makes it possible to reason symbolically about lists built using any associative symbol provided by the user. We have defined a new infix associative symbol `++` with an identity symbol `nil` to represent an event list and also a new auxiliary infix symbol `|>` where the left-hand side contains the mutable memory and the right-hand side contains the event list. The input synchronization message for the button press strand has now the form:

```
{yubikey -> yubikey ;; 1-1 ;;
 Y(pid,sid) @ YubiCounter(pid,c1) @ Server(pid,sid,c2) @ SharedKey(pid,k)
 |> Plugin(pid,c3) ++ Press(pid,c4)}
```

Every time a new event occurs, it is inserted as a new element at the end of the event list. The leftmost elements are the oldest ones, whereas the rightmost elements are the newest. Thus, given an event list L, suppose that we want to verify the property that if event e2 occurs that event e1 occurs before it. We search for cases in which e2 occurs but e1 does *not* occur before it, which means we look for event lists of the form L1 ++ e2 ++ L2, but not L3 ++ e1 ++ L3 ++ e2 ++ L2, where L1, L2, L3, and L4 are variables that can be instantiated to the empty list. If we can show that no such list can be found, we have succeeded the property.

### 4.4. Modeling Lamport clocks

Lamport clocks require the testing of constraints: e.g., whether one counter is smaller than another. This is simple to do when the counters have concrete values. However, since Maude-NPA does not consider concrete protocol states but symbolic state patterns (terms with logical variables), the equality and disequality constraints handled by Maude-NPA are predicates defined over variables, whose domain, in the case of Lamport clocks, is the natural numbers.

In Maude-NPA strands can be extended with equality and disequality constraints [16] of the form "`Term1 eq Term2`" and "`Term1 neq Term2`". Whenever an equality constraint is found during the execution of a strand, the two terms in the equality constraint are unified modulo the equational theory of the protocol and a new state is created for each possible unifier. Whenever a disequality constraint is found during the execution of a strand, it is simply stored in an internal repository of disequality constraints associated to each protocol state; but every time a new state is going to be generated during the state space exploration, all the disequality constraints in the internal repository are tested for satisfiability (see [16] for details).

We deal with Lamport clocks symbolically by representing the relations between clocks as constraints in Presburger Arithmetic. We follow the following encoding of natural numbers. We consider only two constant symbols 0 and 1. Adding two natural numbers $i$ and $j$ is written as $i+j$ where $+$ is an infix associative-commutative symbol with an identity symbol 0. Checking whether a natural number $i$ is smaller than another natural number $j$ is represented in Maude-NPA by a constraint of the form $j$ `eq` $i+k$, where $k$ is an auxiliary variable. Disequality constraints are not needed since the API continues the execution only when the constraints on the Lamport clocks are satisfied.

## 5. Effectiveness of Variant Unification

### 5.1. Evaluation With Respect to the Five Desiderata

We now evaluate variant unification according to the desiderata developed in Section 3.

**des1***: Application to a Large Class of Theories.* In general, variant unification behaves well with respect to this criterion. The major exception is the standard decomposition of the theory describing homomorphic encryption $h$ over a symbol $*$, where $R = \{h(K,X*Y) = h(K,X)*h(K,Y)\}$ and $B$ can be either the empty set of axioms or the $AC$ theory for $*$. In that case, the term $h(K,X)$ has an infinite number of variants [9]:

$$\{(\sigma_0, h(K,X)), (\sigma_1 = \{X \mapsto X_1 * Y_1\}, h(K,X_1)*h(K,Y_1)), \ldots,$$
$$(\sigma_i = \{X_{i-1} \mapsto X_i * Y_i\}, h(K,X*Y)*\ldots h(K,X_1)*h(K,X_2)*\ldots*h(K,X_i)*h(K,Y_i)), \ldots\}$$

In [18], we considered several equational theory decompositions $(\Sigma, B, E)$ for which homomorphic encryption does not have the finite variant property. Thus, unlike the example given in Section 3.2.5, it does not appear to be possible to produce an alternate decomposition with the finite variant property for such theories.

However, there are a number of other theories relevant to cryptographic protocol analysis that include homomorphic encryption equations and do have finite variant decompositions. Even for homomorphic encryption over an AC symbol or a group, it is possible to achieve a sound, but not complete, algorithm by putting a bound on the number of times a homomorphic function symbol can be applied. See [18] for this and other possible ways of handling homomorphic encryption using variant unification.

**des2***: Ease of Verification of Conditions.* We note that some of the six requirements for variant algebraic theories listed in Section 3.2, such as termination, and the finite variant property, are undecidable. But even the undecidable conditions have semi-decision procedure that can be implemented and used to check the conditions. Indeed every single one of these six requirements can be checked by either Maude 2.7.1 [14] or the Maude Formal Environment [19], which require little more from the user than providing the theory to the tool. Thus the work required by the user beyond formulating a theory in the first place is minimal.

**des3***: Combining Disjoint Theories.* For variant unification, the question about combining disjoint finite variant decompositions $(\Sigma_1, B_1, E_1)$ and $(\Sigma_2, B_2, E_2)$ is whether or not $(\Sigma_1 \uplus \Sigma_2, B_1 \uplus B_2, E_1 \uplus E_2)$ has the finite variant property. All that is required is performing the same procedure described in the previous paragraph. Furthermore, some modularity results from the theory of term rewriting can sometimes be used to discharge some of the six requirements.

**des4***: Supporting Irreducibility Constraints.* When performing backwards reachability analysis based on equational unification with a theory $(\Sigma, B, E)$ having the finite variant property, terms describing symbolic states become further and further instantiated by substitutions. But if all the $E,B$-variants of a term $t$ have already been considered and $(u, \sigma)$ is one of them, then we can safely impose some *irreducibility constraint* on $u$ that any further instance $u\rho$ of $u$ should remain $E,B$-irreducible. This is because if $u\rho$ is not so, then the variant $((u\rho)\downarrow_{E,B}, \sigma\rho)$ will be an *instance* of another variant of $u$ corresponding to another symbolic state already explored in the state space.

**des5***: Efficiency.* As expected, variant unification is weakest here. In many cases, it is still efficient enough to be practical, even for theories where $B = AC$. However, there are some important cases in which the number of variants can grow unmanageably large

| | | | | |
|---|---|---|---|---|
| Solution 1 | $X \mapsto Z1 \oplus Z2$ | $Y \mapsto Z2 \oplus Z4$ | $U \mapsto Z1 \oplus Z2$ | $V \mapsto Z3 \oplus Z4$ |
| Solution 2 | $X \mapsto Z1 \oplus Z3$ | $Y \mapsto Z2$ | $U \mapsto Z1 \oplus Z2$ | $V \mapsto Z3$ |
| Solution 3 | $X \mapsto Z1$ | $Y \mapsto Z2 \oplus Z3$ | $U \mapsto Z1 \oplus Z2$ | $V \mapsto Z3$ |
| Solution 4 | $X \mapsto Z1 \oplus Z2$ | $Y \mapsto Z3$ | $U \mapsto Z1$ | $V \mapsto Z2 \oplus Z3$ |
| Solution 5 | $X \mapsto Z1$ | $Y \mapsto Z2$ | $U \mapsto Z1$ | $V \mapsto Z2$ |
| Solution 6 | $X \mapsto Z2$ | $Y \mapsto Z1 \oplus Z3$ | $U \mapsto Z1$ | $V \mapsto Z2 \oplus Z3$ |
| Solution 7 | $X \mapsto Z2$ | $Y \mapsto Z1$ | $U \mapsto Z2$ | $V \mapsto Z2$ |

**Table 2.** Unifiers of $X \oplus Y =_{AC}? U \oplus V$ produced by Maude

very quickly, in particular for $AC$ theories with units and inverses. This includes the $\oplus$ theory we have been using as an example, as well as the Abelian group theories that crop up so often in cryptography. Consider for example the problem of unifying $X \oplus a$ with $Y \oplus b$ presented in Section 3.2.6 which produced a most general set of unifiers with five elements. This does not seem like such a large number, until we realize that there is a most general sets of unifiers of $X \oplus a$ and $Y \oplus b$ with cardinality 1, for example the set $\{X \mapsto Y \oplus a \oplus b\}$. Things become even worse when the number of variables increases. Consider the problem $X \oplus Y =^? U \oplus Z$. According to Table 1, $X \oplus Y$ and $U \oplus V$ have eight variants apiece. Consider the variants $(0, X \oplus Y)$ (equivalent to the first variant produced by Maude in Fig 1), and $(0, Z \oplus Y)$. When we give the problem $X \oplus Y =^?_{AC} U \oplus V$ to Maude, using its built-in $AC$ theory, it produces seven unifiers, as shown in Table 2.

We note that for each unifier $\sigma$, $\sigma X + \sigma Y$ and $\sigma U + \sigma V$ remain irreducible, so all of these are permissible unifiers. Moreover, we still have 48 other pairs of variants to check!

In spite of the tendency to produce large sets of most general unifiers for Abelian group theories, variant unification has been popular in a number of tools for its versatility and ease of application, being used, for example, in the Tamarin tool [20]. Both Maude-NPA and Tamarin have successfully been used to analyze protocols using exclusive-or, e.g. Maude-NPA in [1], and Tamarin in [21]. However, they both can run into state explosion problems once an exclusive-or protocol reaches a certain level of complexity. Thus a unification method that produces fewer unifiers could be useful here.

The key observation is that variant unification is a *theory-generic* unification algorithm applying to an infinite and fairly rich class of equational theories. This gives great flexibility; but it is unrealistic to expect that such a generic algorithm will be able to match the performance of carefully crafted *theory-specific* algorithms. For example, theory-specific algorithms for abelian groups are much more efficient that the generic variant-based unification one. The best approach is of course to exploit the best advantages of the theory-generic and theory-specific algorithms by means of theory combination results [25] that generate a joint unification algorithm for several disjoint theories out of those given for each one. This, however, is non-trivial, because such theory combination methods can easily introduce high levels of non-determinism and may sometimes impose some restrictions on the theories so combined: the devil is in the details. Obtaining an efficient theory combination infrastructure supporting flexible combinations of theory-generic and theory-specific unification algorithms is an important future research goal.

In the next section we will describe efforts to find more efficient theory-generic unification algorithms that still satisfy the desiderata for unification for cryptographic protocol analysis.

## 6. Asymmetric Unification

As we have seen, variant unification, although it has many desirable properties, can in certain cases generate many more variants than we would like. In this section, we will describe some research that has been directed to addressing this problem, namely the study of *asymmetric unification*.

### 6.1. Overview of Asymmetric Unification

The main insight behind asymmetric unification is that variant unification actually overperforms on one of the desiderata: support of irreducibility constraints. In Maude-NPA, and in many other protocol analysis tools, irreducibility is generally required of *negative terms*, i.e., terms corresponding to message reception by a principal or the intruder, in a protocol session or an intruder derivation. This is because the unreachability verification that requires the irreducibility constraint is generally done only for negative terms. Since backwards search is done in Maude-NPA by unifying negative terms received by a principal or intruder with positive terms sent by a principal or intruder, this means that only one of the terms in a unification problem $t =^? s$ has an irreducibility constraint. Thus, the problem we need to solve is to find a most general set of unifiers $\theta$ of $t =^?_{E\cup B} s$ subject to the constraint that $\theta t$ is irreducible, that is $\theta(t)\downarrow_{E,B} =_B t$. Unification with this constraint is called *asymmetric unification* [2] because of the asymmetric nature of the constraint. A methodology for state space exploration using asymmetric unification was presented in [22].

At the time asymmetric unification was first defined, there were no known asymmetric algorithms known other than the one obtained by computing asymmetric unifiers by filtering variant unifiers using variant unification itself. However, in [23] Liu develops an asymmetric algorithm for the Abelian group theory together with free function symbols (that is symbols appearing in no equation), using a methodology for converting symmetric unification algorithms to symmetric ones. A special case of the Abelian group theory, the exclusive-or theory, is presented in [2]. The conversion methodology is given below.

Let $(\Sigma, B, E)$, be a decomposition of an equational theory and let $\Gamma = \{t_1 =\downarrow_{E,B} t'_1, \ldots, t_n =\downarrow_{E,B} t'_n\}$, be an asymmetric unification problem. Then proceed as follows:

1. First compute a complete finite set $S$ of $E\cup B$-unifiers using a finitary unification algorithm for $E\cup B$. If $S$ is empty, then there are no asymmetric unifiers.
2. For each such unifier $\sigma$ from the previous step, check whether every $t'_i\sigma$ is in $E,B$-normal form. All such unifiers are retained also as asymmetric unifiers.
3. For a unifier $\sigma$ such that some $t'_i\sigma$ is not in $E,B$-normal form, compute an equivalent asymmetric unifier if possible.
4. If both of the previous steps fail, this implies that $\sigma$ or its equivalents cannot be asymmetric unifiers in their full generality. However, there may be some instances obtained by instantiating variables in them which are asymmetric unifiers. A complete set of instances of a given unifier is generated by suitably instantiating varibles. This step may be expensive, so it is employed only as a last resort. For each such instance the above steps are repeated.

In Table 3, adapted from [2], we show the results, in terms of the number of unifiers, for asymmetric unification problems encountered in various protocol analyses. We see

| Unif. Problem | #V-U. | V-T. (ms) | #A-U. | A-T. (ms) |
|---:|---:|---:|---:|---:|
| $M_1 \oplus M_2 =\downarrow_{E,B} M_3 \oplus pair(V_1, M_4)$ | 12 | 71 | 1 | 71 |
| $pair(V, rc4(V_1, kAB) \oplus ([N_A, c(N_A)])) =\downarrow_{E,B} pair(V_1, M_1)$ | 1 | 65 | 1 | 70 |
| $M_1 \oplus M_2 =\downarrow_{E,B} M_3 \oplus V_1$ | 12 | 71 | 1 | 71 |
| $M_1 \oplus M_2 =\downarrow_{E,B} M_3 \oplus ([N_1, c(N_2)])$ | 12 | 34 | 1 | 30 |
| $M_1 \oplus M_2 =\downarrow_{E,B} M_3 \oplus pair(V_1, pair(V_2, M_4))$ | 12 | 36 | 1 | 30 |
| $SP4 \wedge SP1 \wedge SP2$ | 4 | 422 | 3 | 68 |
| $SP5 \wedge SP1 \wedge SP2$ | 24 | 408 | 7 | 131 |
| $SP6 \wedge SP1 \wedge SP2$ | 100 | 516 | 15 | 491 |
| $SP7 \wedge SP1 \wedge SP2$ | 360 | 454 | 31 | 3732 |
| $SP8 \wedge SP1 \wedge SP2 \wedge SP3$ | 3 | 151387 | 1 | 47 |
| $SP9 \wedge SP1 \wedge SP2 \wedge SP3$ | 33 | 153913 | 3 | 80 |
| $SP10 \wedge SP1 \wedge SP2 \wedge SP3$ | 201 | 154137 | 7 | 157 |
| $SP11 \wedge SP1 \wedge SP2 \wedge SP3$ | 1053 | 154534 | 15 | 349 |
| $SP12 \wedge SP1 \wedge SP2 \wedge SP3$ | 5073 | 160114 | 31 | 829 |

**Table 3.** Comparisons Between Variant and Asymmetric Unification

#V-U. = # of unifiers using variant unification
V-T. = amount of time in milliseconds using variant unification
SP1 = $M_1 \oplus M_2 =\downarrow_{E,B} M_1 \oplus M_2$
SP2 = $M_1 \oplus M_3 =\downarrow_{E,B} M_1 \oplus M_3$
SP3 = $M_1 \oplus M_4 =\downarrow_{E,B} M_1 \oplus M_4$
SP4 = $M_1 \oplus M_2 \oplus M_3 =\downarrow_{E,B} a \oplus b$
SP5 = $M_1 \oplus M_2 \oplus M_3 =\downarrow_{E,B} a \oplus b \oplus c$
SP6 = $M_1 \oplus M_2 \oplus M_3 =\downarrow_{E,B} a \oplus b \oplus c \oplus d$

#A-U. = # of unifiers using asymmetric unification
A-T. = amount of time in milliseconds using variant unification
SP7 = $M_1 \oplus M_2 \oplus M_3 =\downarrow_{E,B} a \oplus b \oplus c \oplus d \oplus e$
SP8 = $M_1 \oplus M_2 \oplus M_3 \oplus M_4 =\downarrow_{E,B} a$
SP9 = $M_1 \oplus M_2 \oplus M_3 \oplus M_4 =\downarrow_{E,B} a \oplus b$
SP10 = $M_1 \oplus M_2 \oplus M_3 \oplus M_4 =\downarrow_{E,B} a \oplus b \oplus c$
SP11 = $M_1 \oplus M_2 \oplus M_3 \oplus M_4 =\downarrow_{E,B} a \oplus b \oplus c \oplus d$
SP12 = $M_1 \oplus M_2 \oplus M_3 \oplus M_4 =\downarrow_{E,B} a \oplus b \oplus c \oplus d \oplus e$

in particular that for the problem $X \oplus Y =\downarrow_{E,B} U \oplus Z$, variant unification produces 12 unifiers, while an asymmetric unification produces only one. Indeed, in almost all cases, the number of unifiers produced by the asymmetric unification algorithm was smaller than the number or unifiers produced by variant unification, and the advantage increases roughly with the complexity of the unification problem. In addition, with a few exceptions, asymmetric unification times are either similar to or significantly faster than variant unification, especially the last problems involving four equations.

## 6.2. Assessment of Asymmetric Unification

In this section we give an assessment of asymmetric unification with respect to the desiderata for unification algorithms introduced in Section 3.

**des1***: Application to a Large Class of Theories.* In one sense, this condition is trivially satisfied, since it is satisfied by variant unification, from which asymmetric unifiers can be easily derived by a filtering process. However, we are interested in asymmetric unification that improves on the efficiency of variant unification. In that case we know of only one such theory as of now: the Abelian group theory together with uninterpreted function symbols. The question of whether suitable asymmetric unification algorithms can be found for other theories of interest is still an open problem.

**des2***: Ease of Verification of Conditions.* In order to apply asymmetric unification, we need the same conditions to hold as for variant unification. However, the question of whether there are any additional conditions that would make it easier to apply non-variant asymmetric unification is an open problem.

**des3***: Combining Disjoint Theories.* It has been shown by Erbatur et al. in [24] via adaptation of the Baader-Schulz combination method [25] that, given two disjoint equational theories with asymmetric unification algorithms, it is possible to produce a combined asymmetric unification algorithm for the combined theories. Unfortunately, as is true for Baader-Schulz, the problem of determining whether an asymmetric unification problem has a solution in a combined theory is NP-complete. However, there have been optimizations of Baader-Schulz for special cases, some applicable to theories of interest to cryptographic protocol analysis. For example, Tuengerthal [26] has applied optimizations to the Baader-Schulz procedure to develop an algorithm for a combination of the $ACUN$ theory modeling exclusive-or, and a theory modeling pairing, symmetric encryption with cancellation of encryption/decryption, and public key encryption with cancellation of encryption/decryption. It is possible that a similar approach could be applied to combining, say, variant unification for cancellation rules with the Abelian group algorithm developed in [23], to give us an asymmetric algorithm that supports a combination of Abelian groups, uninterpreted function symbols, and cancellation rules.

**des4***: Supporting Irreducibility Constraints.* Asymmetric unification supports irreducibility constraints easily. It is possible to enforce a constraint on any term $t$, even if it does not originally appear in the unification problem, by adding the equation $t =\downarrow_{E,B} t$ to the problem. This is done, for example, in several of the problems presented in Table 3, such as the problem $SP4 \wedge SP1 \wedge SP2$.

**des5***: Efficiency.* As we have seen, it is possible to greatly reduce the number of unifiers over variant unification for exclusive-or with uninterpreted function symbols. For the problems in Table 3, the asymmetric algorithm produced fewer unifiers than the variant algorithm, sometimes significantly fewer. In terms of time required to perform the unification, the asymmetric algorithm performed similarly to or significantly better than the variant algorithm, except for one case in which the variant algorithm performed significantly better.

## 7. Conclusion

We have presented an overview of Maude-NPA and of the way it uses unification to perform backwards search. We have used this to motivate a study of the type of equational unification used in Maude-NPA. We have then given an in-depth presentation of the equational unification algorithm used by Maude-NPA: variant unification. We also presented an evaluation of variant unification from the point of view of cryptographic protocol verification. We have used that to motivate a study of a possible enhancement of variant unification, asymmetric unification, and of how asymmetric unification could enhance both variant-unification and theory-specific algorithms. We used the same desiderata used to evaluate variant unification to help point out where research was most needed in asymmetric unification.

We note that, although we have restricted our discussion to unification as it is used in Maude-NPA, this work has potential applications to other tools as well. As we pointed out earlier, variant unification is employed by a number of cryptographic protocol analysis tools. We hope that this presentation will stimulate research in this area, and lead to improved unification algorithms for cryptographic protocol analysis.

## References

[1] Antonio González-Burgueño, Damián Aparicio-Sánchez, Santiago Escobar, Catherine A. Meadows, and José Meseguer. Formal verification of the YubiKey and YubiHSM APIs in Maude-NPA. In *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16-21 November 2018*. EPiC Series in Computing 57, pages 400–417, EasyChair 2018.

[2] Serdar Erbatur, Santiago Escobar, Deepak Kapur, Zhiqiang Liu, Christopher Lynch, Catherine A. Meadows, José Meseguer, Paliath Narendran, Sonia Santiago, and Ralf Sasse. Asymmetric unification: A new unification paradigm for cryptographic protocol analysis. In *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, pages 231–248, 2013.

[3] N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 243–320. Elsevier, Amsterdam, 1990.

[4] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[5] J.-P. Jouannaud, C. Kirchner, and H. Kirchner. Incremental construction of unification algorithms in equational theories. In *Proc. ICALP'83*, pages 361–373. Springer LNCS 154, 1983.

[6] Santiago Escobar, Ralf Sasse, and José Meseguer. Folding variant narrowing and optimal variant termination. *The Journal of Logic and Algebraic Programming*, 81(7-8):898–928, 2012.

[7] Andrew Cholewa, Jose Meseguer, and Santiago Escobar. Variants of variants and the finite variant property. Technical report, University of Illinois at Urbana-Champaign, http://hdl.handle.net/2142/47117, 2014.

[8] José Meseguer. Strict coherence of conditional rewriting modulo axioms. *Theor. Comput. Sci.*, 672:1–35, 2017.

[9] Hubert Comon-Lundh and Stéphanie Delaune. The finite variant property: How to get rid of some algebraic properties. In *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings*, pages 294–307, 2005.

[10] Yubico. YubiHSM Manual v1.5. Available on: https://www.yubico.com/wp-content/uploads/2015/04/YubiHSM-Manual_1_5_0.pdf.

[11] Robert Künnemann and Graham Steel. YubiSecure? formal security analysis results for the Yubikey and YubiHSM. In *Revised Selected Papers of the 8th Workshop on Security and Trust Management (STM'12)*, volume 7783 of Lecture Notes in Computer Science, pages 257272, Pisa, Italy, September 2012. Springer.

[12] Robert Künnemann. *Foundations for analyzing security APIs in the symbolic and computational model. Available on: https://tel.archives-ouvertes.fr/tel-00942459/file/Kunnemann2014.pdf.* Theses, École normale supérieure de Cachan - ENS Cachan, January 2014.

[13] Sonia Santiago, Santiago Escobar, Catherine A. Meadows, and José Meseguer. Effective sequential protocol composition in Maude-NPA. CoRR, abs/1603.00087, 2016.

[14] Maude 2.7.1. Available at http://maude.cs.illinois.edu/w/index.php/The_Maude_System.

[15] Francisco Durán, Steven Eker, Santiago Escobar, Narciso Martí-Oliet, José Meseguer, Rubén Rubio, and Carolyn Talcott. Programming and Symbolic Computation in Maude. *Journal of Logical and Algebraic Methods in Programming*, to appear, 2019.

[16] Santiago Escobar, Catherine A. Meadows, José Meseguer, and Sonia Santiago. Symbolic protocol analysis with disequality constraints modulo equational theories. In *Programming Languages with Applications to Biology and Security - Essays Dedicated to Pierpaolo Degano on the Occasion of His 65th Birthday*, volume 9465 of Lecture Notes in Computer Science, pages 238261. Springer, 2015.

[17] Christopher Bouchard, Kimberly A. Gero, Christopher Lynch, and Paliath Narendran. On forward closure and the finite variant property. In *Frontiers of Combining Systems - 9th International Symposium, FroCoS 2013, Nancy, France, September 18-20, 2013. Proceedings*, pages 327–342, 2013.

[18] Fan Yang, Santiago Escobar, Catherine A. Meadows, Jos Meseguer, Paliath Narendran. Theories of Homomorphic Encryption, Unification, and the Finite Variant Property. In *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming* , pages 123-133, 2014.

[19] The Maude Formal Environment. Available at https://code.google.com/archive/p/maude-formal-environment/.

[20] Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. The TAMARIN prover for the symbolic analysis of security protocols. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 696–701, 2013.

[21] Jannik Dreier, Lucca Hirschi, Sasa Radomirovic, and Ralf Sasse. Automated unbounded verification of stateful cryptographic protocols with exclusive OR. In *31st IEEE Computer Security Foundations Symposium, CSF 2018*, pages 359–373, 2018.

[22] Serdar Erbatur, Santiago Escobar, Deepak Kapur, Zhiqiang Liu, Christopher Lynch, Catherine A. Meadows, José Meseguer, Paliath Narendran, Sonia Santiago, and Ralf Sasse. Effective symbolic protocol analysis via equational irreducibility conditions. In Sara Foresti, Moti Yung, and Fabio Martinelli, editors, *Computer Security - ESORICS 2012 - 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings*, volume 7459 of *Lecture Notes in Computer Science*, pages 73–90. Springer, 2012.

[23] Zhiqiang Liu. *Dealing Efficiently with Exclusive Or, Abelian Groups and Homomorphism in Cryptographic Protocol Analysis*. PhD thesis, Clarkson University, Available at http://people.clarkson.edu/?clynch/papers/DissertationofZhiqiangLiu.pdf, 2012.

[24] Serdar Erbatur, Deepak Kapur, Andrew M. Marshall, Catherine A. Meadows, Paliath Narendran, and Christophe Ringeissen. On asymmetric unification and the combination problem in disjoint theories. In *Proceedings of FOSSACS 2014*, pages 274–288, 2014.

[25] Franz Baader and Klaus U. Schulz. Unification in the union of disjoint equational theories: Combining decision procedures. In *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction, Saratoga Springs, NY, USA, June 15-18, 1992, Proceedings*, pages 50–65, 1992.

[26] Max Tuengerthal. Implementing a unification algorithm for protocol analysis with XOR. Technical Report 0609, Institut für Informatik, CAU Kiel, Available at https://sec.uni-stuttgart.de/_media/publications/Tuengerthal-IFI-TR-0609-2006.pdf, 2006.

*Engineering Secure and Dependable Software Systems*
*A. Pretschner et al. (Eds.)*
*IOS Press, 2019*
*© 2019 The authors and IOS Press. All rights reserved.*
*doi:10.3233/978-1-61499-977-5-189*

# Building Deductive Program Verifiers

*Lecture Notes*

Peter MÜLLER[a]
[a] *Department of Computer Science, ETH Zurich, Switzerland*

**Abstract.** Deductive program verifiers attempt to construct a proof that a given program satisfies a given specification. Their implementations reflect the semantics of the programming language and the specification language, and often include elaborate proof search strategies to automate verification. Each of these components is intricate, which makes building a verifier from scratch complex and costly.

In these lecture notes, we will present an approach to build program verifiers as a sequence of translations from the source language and specification via intermediate languages down to a logic for which automatic solvers exist. This architecture reduces the overall complexity by dividing the verification process into simpler, well-defined tasks, and enables the reuse of essential elements of a program verifier such as parts of the proof search, specification inference, and counterexample generation. We will use the intermediate verification language Viper to demonstrate how to encode interesting verification problems.

**Keywords.** verification condition generation, intermediate verification language, permission logics, hyperproperties, product programs, Nagini, Viper

## 1. Introduction

Ensuring the correctness and security of software systems is becoming increasingly challenging. Testing has always been limited to checking just a small subset of the possible program executions. In the omnipresence of concurrency (for instance, in software that runs on multicore processors or in data centers) and event-based systems (such as applications running on mobile devices), testing is largely insufficient. It is, thus, useful to complement or replace testing with static verification techniques such as static program analysis [9], model checking [7], or deductive verification [19]. These techniques can formally prove correctness and security properties for all executions of a program, that is, for all possible inputs, thread schedules, event interactions, attacker behaviors, etc.

Static program analysis, model checking, and deductive verification strike different trade-offs between automation, expressiveness, and modularity. In these lecture notes, we focus on deductive verification, which requires more user input than the other techniques, but allows one to prove complex properties and enables modular verification [28]. Modularity is important for scalability, to reduce the re-verification effort during software maintenance, and to give guarantees for individual program components such as libraries. Deductive program verification employs a program logic such as Hoare logic [19] or separation logic [33] to construct a mathematical proof that a given program satisfies its specification.

*Program verifiers* are tools that automate (parts of) the proof search, typically by reducing verification to a set of verification conditions, logical formulas whose validity implies the correctness of the program and which can be checked by automatic or interactive theorem provers. This reduction is often performed by encoding a program, its specification, and the program logic into an intermediate verification language. Programs in the intermediate language are typically not (efficiently) executable. Their correctness implies the correctness of the original program. Implementing a verifier via an intermediate verification language has two major advantages over a monolithic architecture. First, it allows one to reuse large parts of the tool infrastructure; all components that operate on the intermediate language or further downstream can be reused across multiple program verifiers just like an optimizer and a code generator for an compiler intermediate language can be reused across multiple compilers. Second, human-readable intermediate verification languages greatly simplify the prototyping of verification techniques and tools, as well as debugging.

There are several mature intermediate verification languages and corresponding tool infrastructures. Boogie [22] offers a simple procedural language and tool support for verification condition generation, bounded verification [20], and debugging of verification failures [21]. Why's language [16] has a functional flavor; its verifier targets a wide range of automatic provers. Viper [30] facilitates the verification of proofs in logics similar to separation logic, targeting especially heap-manipulating and concurrent programs. All three intermediate languages are widely used. For instance, Boogie is at the core of verifiers such as Chalice [26], Corral [20], Dafny [23], Spec# [25,2], and VCC [8]. Why powers for instance Frama-C and Krakatoa [15], and Viper is used by Nagini [11], Prusti [1], and VerCors [5].

In these lecture notes, we will use Viper. However, especially the concepts introduced in the earlier sections apply similarly to other intermediate verification languages. Fig. 1 shows the architecture of the Viper verification infrastructure. We will introduce the Viper intermediate language together with examples later, but refer to an overview of its design [30] and the tutorial [13] for details. Viper provides two backend verifiers. The symbolic execution verifier [35] reasons about heap manipulations internally and uses the SMT solver Z3 [27] for other aspects of verification, such as arithmetic. The verification condition generator is itself implemented via a translation to the intermediate language Boogie, which ultimately also targets Z3. Both tools can be tried online at `viper.ethz.ch`; the entire infrastructure is available as open-source implementation.

*Outline.* Sec. 2 introduces the foundations of verification condition generation. Sec. 3 shows how to verify hyperproperties (properties the relate two or more program executions) via an encoding onto an intermediate language. In Sec. 4, we explain how to encode the verification of heap-manipulating programs using a flavor of separation logic. Sec. 5 extends this encoding to recursive data structures. We briefly discuss the development of frontend verifiers in Sec. 6, and conclude in Sec. 7.

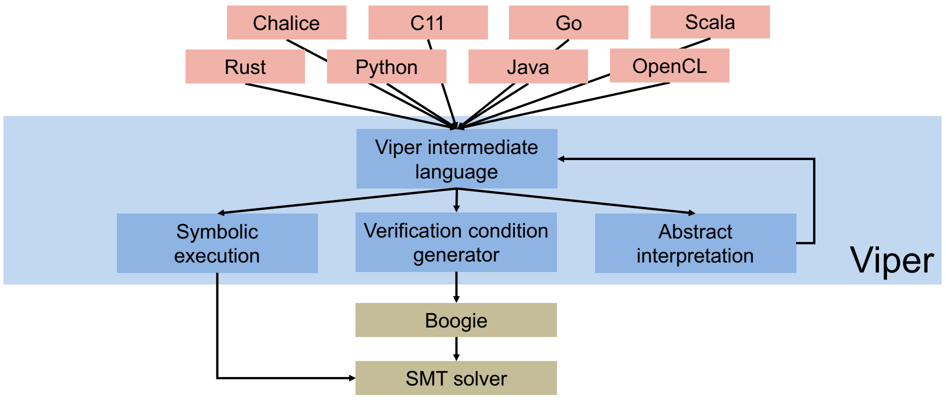


**Figure 1.** Architecture of the Viper verification infrastructure. The blue boxes depict the main components of the architecture: the Viper intermediate language, two verification backends (based on symbolic execution and verification condition generation, resp.), and an abstract interpreter to infer auxiliary specifications. Gray boxes are external components, and orange boxes show the languages for which existing verifiers are implemented via a translation into Viper. The verifiers in the upper row (Chalice, C11, Go, Scala) are prototypes, whereas the others (Rust, Python, Java, OpenCL) are fairly mature tools.

## 2. Verification Condition Generation

Throughout these lecture notes, we will encode more and more complex programs and properties into simpler intermediate representations. As foundation for this chain of translations, we employ Dijkstra's guarded-commands language [10]. In this section, we introduce the necessary background of guarded commands, explain how to encode statements into this language, and illustrate the resulting verification approach on an example.

### *2.1. Guarded Commands*

We use a guarded-commands language with the following syntax:

$$\begin{array}{rl} S ::= & x{:=}e \\ | & \texttt{havoc}\ x \\ | & \texttt{assert}\ P \\ | & \texttt{assume}\ P \\ | & S;S \\ | & S \mathbin{[\!]} S \end{array}$$

where $x$ ranges over variables, $e$ denotes a side-effect free expression, and $P$ denotes an assertion (a first-order formula over program variables). Guarded commands include assignments, assignments of non-deterministic values to variables, assertions, assumptions, sequential composition, and non-deterministic choice. The execution of a guarded command from an initial state fails if the condition of an assertion evaluates to false, it is infeasible if the condition of an assumption evaluates to false, and otherwise succeeds. We say that a guarded command is *correct* if all feasible executions succeed, that is, no execution fails.

Correctness of a guarded command $S$ can be verified by proving the validity of the verification condition $wp(S, true)$, where $wp(S,Q)$ denotes the weakest precondition of guarded command $S$ w.r.t. assertion $Q$ and is defined as follows ($Q[e/x]$ denotes $Q$ with $e$ substituted for $x$):

$$\begin{aligned}
wp(x{:=}e, Q) &= Q[e/x] \\
wp(\texttt{havoc } x, Q) &= \forall x \cdot Q \\
wp(\texttt{assert } P, Q) &= P \wedge Q \\
wp(\texttt{assume } P, Q) &= P \Rightarrow Q \\
wp(S_1; S_2, Q) &= wp(S_1, wp(S_2, Q)) \\
wp(S_1 \mathbin{[\!]} S_2, Q) &= wp(S_1, Q) \wedge wp(S_2, Q)
\end{aligned}$$

Many verification problems require mathematical theories such as arithmetic and set theory. For those theories that are not natively supported by the underlying theorem prover, intermediate verification languages allow programmers to define them via sorts, uninterpreted function symbols, and axioms. The conjunction of these axioms forms a so-called *background predicate*, which may be assumed for any given verification task. So for a background predicate *BP*, the verification of a guarded command $S$ means proving the validity of:

$$BP \Rightarrow wp(S, true)$$

## *2.2. Encoding of Statements*

Guarded commands offer a simple core language, for which verification condition generation is straightforward. Other statements can be encoded conveniently into guarded commands, as we show next.

*Conditional statements.* A conditional statement

$$\texttt{if } e \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ end}$$

is encoded as

$$(\texttt{assume } e;\ [\![S_1]\!]) \mathbin{[\!]} (\texttt{assume } \neg e;\ [\![S_2]\!])$$

where $[\![S]\!]$ denotes the encoding of statement $S$ (we omit the encoding of expressions for simplicity). Intuitively, the resulting guarded command is correct if $S_1$ is correct *if* $e$ holds (otherwise the left-hand side of the non-deterministic choice is infeasible) and if $S_2$ is correct *if* $e$ does not hold (otherwise the right-hand side is infeasible). This is exactly the verification condition required for a conditional statement, and formalized by the verification condition:

$$(e \Rightarrow wp([\![S_1]\!], Q)) \wedge (\neg e \Rightarrow wp([\![S_2]\!], Q))$$

*Loops.* The verification of loops requires a suitable loop invariant, that is, an assertion that holds before the loop and after each loop iteration. A loop invariant represents the inductive argument needed to reason about an unknown number of loop iterations. We require programmers to provide loop invariants manually through suitable annotations in the code. However, techniques for the automatic inference of invariants, typically via fixpoint iteration, exist [9].

A loop of the form

```
while($e$) invariant $P$ begin $S$ end
```

is encoded as

```
assert $P$;
havoc $x_i$; assume $P$;
(assume $e$; $[\![S]\!]$; assert $P$; assume false)
$[\!]$
assume $\neg e$
```

This encoding first asserts the loop invariant before the loop. Instead of performing an iteration (which would require a fixpoint computation in the *wp* computation), it simulates an arbitrary loop execution. For this purpose, we assign non-deterministic values to all variables $x_i$ that are assigned in the loop body $S$ (the so-called loop targets) and assume the loop invariant. This step removes all previous knowledge about these variables, except that they satisfy the loop invariant. The subsequent non-deterministic choice models the two possible behaviors of the loop. If the loop condition holds, we execute the loop body $S$ and prove that it preserves the invariant $P$. The subsequent `assume false` ensures that any subsequent code verifies trivially; this is needed because we encode the termination of the loop separately, in the second branch of the non-deterministic choice. Here, we assume that the loop condition $e$ does not hold.

Verifying the encoding of a loop ensures that the loop is correct if it terminates. Proof obligations that enforce termination need to be encoded explicitly, as we discuss later.

*Procedures.* Modular verification techniques require specifications for procedures and verify calls using the specification of the callee instead of its implementation. This approach is compatible with information hiding, allows one to verify calls where the callee implementation is unknown (for instance, abstract methods, dynamically-bound methods, or library methods), and avoids re-verification of callers when the implementation of a callee changes.

Procedures are typically specified using pre- and postconditions. Callers need to establish the precondition and may assume the postcondition after the call. In turn, procedure bodies may assume that their precondition holds upon entry and must establish the postcondition upon termination. Consider a procedure declaration

```
procedure $p(x)$ returns $r$
  requires $P$
  ensures $Q$
begin $S$ end
```

where $x$ is the (only) formal parameter, $r$ is the result variable, $P$ is the precondition, and $Q$ is the postcondition. In languages without global state such as global variables or a heap memory, correctness of this procedure can be encoded into guarded commands as follows:

```
assume $P$
$[\![S]\!]$
assert $Q$
```

A a call $y$ `:=` $p(e)$ is encoded as (where $z$ is a fresh variable):

```
$z$ := $e$
assert $P[z/x]$
havoc $y$
assume $Q[z/x, y/r]$
```

The substitutions replace the parameter and result variable in the pre- and postcondition by the actual argument and right-hand side variable, resp. The temporary variable $z$ is needed since $e$ may refer to $y$. The havoc operation reflects that the call updates variable $y$ and, thus, all prior information about its value is no longer valid. Properties of the new value of $y$ are conveyed via $p$'s postcondition. We will discuss in Sec. 4 how to encode procedures and calls in the presence of a heap memory, where we need to reflect the potential side effects of a call on heap locations.

### *2.3. Example*

The example in Fig. 2 illustrates the concepts used so far. This Viper procedure (called *method* in Viper) implements the first challenge of the VerifyThis 2011 verification competition (see `www.pm.inf.ethz.ch/research/verifythis/Archive/2011.html`). Method `maxSeq` computes the index of the maximum of a non-empty sequence of integers. The postcondition states that the result `x` is a valid index and that the value at position `x` is at least as large as all other values in the sequence. The loop invariant states that values to the left of `x` and to the right of `y` are less than or equal to the value at position `x`. Consequently, when the loop terminates, we have `x==y` and, thus, `x` is the index of the maximum.

`Seq` is a built-in generic datatype. It is encoded via uninterpreted function symbols and axioms, which are part of the background predicate used to verify any Viper method.

Viper does not require termination of methods and loops. However, it is possible to encode termination arguments explicitly via additional assertions in the code. In this example, we use `y - x` as ranking function. To ensure that its value ranges over a well-founded set, we prove in line 18 that it is non-negative. Termination is then guaranteed by the fact that each loop iteration decreases the value of the ranking function, which we assert in line 25.

## 3. Verification of Hyperproperties

With the technique introduced so far, we can prove properties for individual executions of a program such as functional correctness and termination. Other im-

```
method maxSeq(s: Seq[Int]) returns (x: Int)
  requires 0 < |s|
  ensures 0 <= x && x < |s|
  ensures forall i: Int :: 0 <= i && i < |s| ==> s[i] <= s[x]
{
  x := 0
  var y: Int := |s| - 1

  while(x != y)
    invariant 0 <= x
    invariant 0 <= y && y < |s|
    invariant x <= y
    invariant forall i: Int ::
                    (0 <= i && i < x || y < i && i < |s|)
                    ==> (s[i] <= s[x] || s[i] <= s[y])
  {
    var measure: Int  := y - x   // termination
    assert 0 <= measure          // termination
    if(s[x] <= s[y])
    {
      x := x + 1
    } else {
      y := y - 1
    }
    assert y - x < measure       // termination
  }
}
```

**Figure 2.** A Viper method that computes the index of the maximum of a non-empty sequence of integers. The pre- and postcondition express the functional behavior of the method; termination is encoded manually through local assertions.

portant properties relate multiple executions. For instance, determinism requires that two executions starting from the same initial state terminate in the same final state. Properties of multiple program executions are called *hyperproperties* and include for instance monotonicity, non-interference [17] (which is used to prove secure information flow [34]), or read effects [24].

Hyperproperties can be verified via relational program logics [4,39]. However, these logics are difficult to automate and require dedicated tool support. An alternative is to construct a so-called *product program* [3,12] that encodes two or more executions of the original program into a single execution of the product. This product program can be expressed in an intermediate verification language and verified using the approach introduced above.

We consider *modular product programs* [12] here, which enable the modular verification of hyperproperties, and focus on the encoding of two executions. A generalization to arbitrary numbers is trivial. The basic construction is simple. To encode the state space of two program executions, we introduce two variables $x_1$ and $x_2$ for each variable $x$ of the original program. Since the control flow of the two executions of the original program may differ, we introduce two boolean *activation* *variables* $p_1$ and $p_2$ that reflect whether each of the executions is currently active. Both variables are initially true.

An assignment $x$:=$e$ in the original program is then encoded as two conditional assignments, which update the variables of the product program *if* the corresponding execution is active:

```
if($p_1$) { $x_1$:=$e_1$ }
if($p_2$) { $x_2$:=$e_2$ }
```

where $e_i$ is the expression $e$ with all occurrences of a variable $y$ replaced by $y_i$. A conditional statement `if(`$e$`) {` $S_1$ `} else {` $S_2$ `}` is encoded by introducing fresh activation variables that reflect which execution enters the then- and the else-branch, resp.:

```
var $p_1^{true}$ := $p_1 \wedge e_1$
var $p_2^{true}$ := $p_2 \wedge e_2$
var $p_1^{false}$ := $p_1 \wedge \neg e_1$
var $p_2^{false}$ := $p_2 \wedge \neg e_2$
$[\![S_1]\!](p_1^{true}, p_2^{true})$
$[\![S_2]\!](p_1^{false}, p_2^{false})$
```

where $[\![S]\!](p_1, p_2)$ denotes the product construction for statement $S$ with activation variables $p_1$ and $p_2$.

A key feature of modular product programs is that their construction *does not* duplicate loops and method calls. This feature allows us to use loop invariants and method specifications that relate both executions of the original program and, thereby, enables modular verification. A loop `while(`$e$`) invariant` $P$ `{` $S$ `}` is encoded as follows:

```
while($p_1 \wedge e_1 \vee p_2 \wedge e_2$)
  invariant $[\![P]\!](p_1, p_2)$
{
  var $p_1^{true}$ := $p_1 \wedge e_1$
  var $p_2^{true}$ := $p_2 \wedge e_2$
  $[\![S]\!](p_1^{true}, p_2^{true})$
}
```

The product loop iterates as long as one of the two executions is active and its loop condition is satisfied. However, the loop body is executed only for active executions.

Modular product programs support both classical and relational assertions. A classical assertion $P$ is encoded as $(p_1 \Rightarrow P_1) \wedge (p_2 \Rightarrow P_2)$, that is, it must hold for each active execution. Relational specifications may relate both executions of the program. A relational assertion $R$ is encoded as $p_1 \wedge p_2 \Rightarrow R$, that is, it must hold if both executions are active (the variables of an inactive execution do not have meaningful values).

Methods are encoded by duplicating parameters and results, and adding two extra parameters for the activation variables. A call then passes the values of the

```
method maxDet(s1: Seq[Int], s2: Seq[Int], p1: Bool, p2: Bool)
                                         returns (x1: Int, x2: Int)
  requires p1 ==> 0 < |s1|
  requires p2 ==> 0 < |s2|
  ensures p1 && p2 ==> (s1 == s2 ==> x1 == x2)
{
  var y1: Int
  var y2: Int
  if(p1) { x1 := 0 }
  if(p2) { x2 := 0 }
  if(p1) { y1 := |s1| - 1 }
  if(p2) { y2 := |s2| - 1 }

  while(p1 && x1 != y1 || p2 && x2 != y2)
    invariant p1 ==> 0 <= x1 && 0 <= y1 && y1 < |s1| && x1 <= y1
    invariant p2 ==> 0 <= x2 && 0 <= y2 && y2 < |s2| && x2 <= y2
    invariant p1 && p2 ==> (s1 == s2 ==> x1 == x2 && y1 == y2)
  {
    var pw1: Bool := p1 && x1 != y1
    var pw2: Bool := p2 && x2 != y2

    var pt1: Bool := pw1 && s1[x1] <= s1[y1]
    var pt2: Bool := pw2 && s2[x2] <= s2[y2]
    var pf1: Bool := pw1 && !(s1[x1] <= s1[y1])
    var pf2: Bool := pw2 && !(s2[x2] <= s2[y2])

    if(pt1) { x1 := x1 + 1 }
    if(pt2) { x2 := x2 + 1 }
    if(pf1) { y1 := y1 - 1 }
    if(pf2) { y2 := y2 - 1 }
  }
}
```

**Figure 3.** Modular product program for the method from Fig. 2. The classical preconditions ensure that sequence accesses are within bounds. The relational postcondition expresses determinism: for equal parameter values, we will get equal results. We omit the termination checks for simplicity.

caller's activation variables to the callee to ensure that the body of the callee method is executed only if the corresponding execution is active.

Fig. 3 shows the product program for the example in Fig. 2. The classical preconditions (and the corresponding loop invariants) ensure that sequence accesses are within bounds. The relational postcondition and loop invariant express determinism: for equal parameter values, we will get equal results in both method executions.

Modular product programs allow one to use off-the-shelf verifiers to verify hyperproperties. They can be used to verify even advanced non-interference properties including declassification and the absence of termination leaks [12].

## 4. Verification of Heap-Manipulating Programs

In this section, we present a verification technique for heap-manipulating programs and show how it can be encoded into the language introduced so far.

### 4.1. Access Permissions

The main verification challenge for heap-manipulating programs is *framing*: how to preserve information about heap data structures across heap changes, in particular, across method calls. For this purpose, we introduce the notion of an *access permission* (or permission for short), which facilitates static verification, but is not present during program execution. We associate an access permission with each heap location. This permission is created when the heap location is allocated. Permissions are held by method executions; a method execution may access a heap location only if it holds the corresponding permission. Permissions may be transferred between method executions, but cannot be duplicated or forged. Consequently, there is at most one permission available for each location. While one method holds the permission, no other method can have it to modify the location, which enables framing.

To distinguish read and write accesses, it is useful to support *fractional permissions* [6], where a permission can be split into several fractions, and the fractions can be re-combined to obtain a full permission. Writing to a memory location requires full permission, whereas any non-zero fraction permits reading.

Let us assume a heap that consists of objects with fields. We can encode heaps and permissions as two mathematical maps (defined as part of the background predicate) that map reference-field pairs to values and rational numbers in $[0;1]$, resp. We call the permission map a *mask*. Using this encoding, a field read $y$ `:=` $x.f$ is encoded as:

```
assert $x \neq null$
assert $Mask[x,f] > 0$
$y$ := $Heap[x,f]$
```

Analogously, we can encode a field update $x.f$ `:=` $e$ as:

```
assert $x \neq null$
assert $Mask[x,f] = 1$
$Heap$ := $Heap[x,f \to e]$
```

Permissions are transferred between methods upon calls and when a call returns. Which permissions to transfer is specified in the method specification via accessibility predicates of the form `acc(`$x.f$`, `$p$`)`, where $p$ is the required fraction. The transfer is encoded via two auxiliary operations on assertions. *Exhaling* an assertion $P$ is done in three steps: (1) It asserts that all permissions required by $P$ are available in the current mask and that all logical constraints in $P$ hold; if not, verification fails. (2) It removes the transferred permissions from the mask. (3) It havocs all memory locations to which no permission is held, to reflect that other methods may use the permission to update those locations and, thereby, invalidate any knowledge about them. Conversely, *inhaling* an assertion $P$ requires

```
field val: Int
define read(a,i)
  slot(a,i).val

domain IArray {
  function slot(a: IArray, i: Int): Ref
  function len(a: IArray): Int
  function first(r: Ref): IArray
  function second(r: Ref): Int

  axiom all_diff {
    forall a: IArray, i: Int :: { slot(a,i) }
      first(slot(a,i)) == a && second(slot(a,i)) == i
  }

  axiom len_nonneg {
    forall a: IArray :: { len(a) }
      len(a) >= 0
  }
}
```

**Figure 4.** A Viper field, macro, and background predicate to encode mutable integer arrays. The term `{ slot(a,i) }` is a matching pattern used by the SMT solver to instantiate the universal quantifier.

two steps: (1) It adds the transferred permissions to the mask. (2) It assumes that all logical constraints in $P$ hold. Both operations are defined inductively over the syntax of assertions; we omit a formal encoding here for simplicity.

With these auxiliary operations, we can adapt the encoding of procedure declarations and calls presented earlier. Instead of asserting and assuming pre- and postconditions, they are exhaled and inhaled, resp. For instance, upon a call, the caller exhales the precondition (to transfer permissions to the callee) and then inhales the postcondition (to transfer permissions back). The havoc that happens in step 3 of the exhale reflects the potential side effects of the callee method.

Permissions are the basis behind modern program logics such as separation logic [33] and implicit dynamic frames [36]. They are applicable to a wide range of verification problems. In a concurrent setting, they ensure data race freedom [31]. If one thread holds full permission to write to a memory location, other threads hold no permission and can, thus, neither read nor write. Nevertheless, fractional permissions enable concurrent reading. Permissions have also been used to verify fine-grained concurrency, even on weak memory models [38,37].

### *4.2. Example*

To illustrate the use of permissions, we discuss a variation of the example from Fig. 2 that operates on a mutable array instead of a mathematical sequence.

Viper does not support arrays natively, but they can be encoded easily as part of the background predicate, as shown in Fig. 4. We model each array location as

```
method maxArray(a: IArray) returns (x: Int)
  requires 0 < len(a)
  requires forall i: Int :: 0<=i && i<len(a) ==> acc(read(a,i), 1/2)
  ensures 0 <= x && x < len(a)
  ensures forall i: Int :: 0<=i && i<len(a) ==> acc(read(a,i), 1/2)
  ensures forall i: Int :: 0<=i && i<len(a)
          ==> read(a,i) <= read(a,x)
{
  x := 0
  var y: Int := len(a) - 1

  while(x != y)
    invariant 0 <= x && x <= y && y < len(a)
    invariant forall i: Int :: 0<=i && i<len(a)
              ==> acc(read(a,i), 1/2)
    invariant forall i: Int :: (0<=i && i<x || y<i && i<len(a))
              ==> (read(a,i) <= read(a,x) || read(a,i) <= read(a,y))
  {
    var measure: Int := y - x    // termination
    assert 0 <= measure          // termination
    if(read(a,x) <= read(a,y))
    {
      x := x + 1
    } else {
      y := y - 1
    }
    assert y - x < measure       // termination
  }
}
```

**Figure 5.** A variation of the example from Fig. 2 that operates on a mutable array instead of a mathematical sequence. The fractional permissions in the method specification and loop invariant allow the method to read the array elements, but prevent modifications. They allow callers to conclude that the array is not changed by the method.

a separate reference, which is yielded by function `slot`. The value of this location can then be accessed via a predefined `val` field of that reference. To simplify the notation, we define a macro `read` that is parametric in the array and index, and expands into the access expression. The first axiom states that `slot` is injective. It is expressed via two inverse functions, which yield better performance in the SMT solver than a naive formulation of injectivity.

Fig. 5 shows the Viper encoding of the example. The fractional permissions in the method specification and loop invariant allow the method to read the array elements, but prevent modifications. Therefore, they enable framing: callers may conclude that the array is not changed by the method. The assertions use universal quantification over permissions, a feature called *iterated separating conjunction* [33], which is supported by Viper [29].

```
field left: Ref
field right: Ref
field val: Int

predicate tree(this: Ref) {
  acc(this.left) && acc(this.right) && acc(this.val) &&
  (this.left != null ==> tree(this.left)) &&
  (this.right != null ==> tree(this.right))
}

function elems(this: Ref): Multiset[Int]
  requires tree(this)
{
  unfolding tree(this) in
    Multiset(this.val) union
    (this.left != null ? elems(this.left) : Multiset[Int]()) union
    (this.right != null ? elems(this.right) : Multiset[Int]())
}
```

**Figure 6.** A recursive tree predicate and a heap-dependent function to obtain the multiset of integers stored in a tree.

## 5. Verification of Recursive Data Structures

Iterated separating conjunction lets us specify permissions to a statically-unknown number of memory locations. It is especially useful for random-access data structures such as arrays and maps. For other data structures, in particular recursive ones, we can specify permissions using (possibly recursive) predicates [32,18].

The predicate `tree` in Fig. 6 provides permission to the fields of the argument reference. If either of the two subtrees is non-null, it includes a recursive predicate instance for this tree. Consequently, the predicate represents the permissions to all locations in the entire tree. A predicate may also constrain the values of heap locations whose permissions it contains, for instance, to express invariants of data structures.

Just like permissions, predicate instances are held by method executions and transferred through exhale and inhale operations. Predicates with one parameter such as the `tree` predicate can be stored in the mask. For instance, $Mask[x,P] = 1$ expresses that the current state contains the predicate $P(x)$. Predicates with more arguments require higher-dimensional masks.

Recursive definitions are generally tricky for automatic provers because provers need to be prevented from unfolding them indefinitely, leading to non-termination in the proof search. To avoid this problem, many tools require programmers to unfold and fold predicates manually through annotations in the code. For this purpose, Viper provides a statement `unfold` $P(x)$, which exhales the predicate instance $P(x)$ and inhales the body of the predicate. Conversely, `fold` $P(x)$ exhales the body and inhales the predicate instance. an expression `unfolding` $P(x)$ `in` $e$ temporarily unfolds $P(x)$, evaluates expression $e$, and then re-folds $P(x)$.

```
 1  method maxTree(t: Ref) returns (m: Int)
 2    requires tree(t)
 3    ensures tree(t)
 4    ensures 0 < (m in elems(t))
 5    ensures forall e: Int :: 0 < (e in elems(t)) ==> e <= m
 6    ensures elems(t) == old(elems(t))
 7  {
 8    var tmp: Int
 9    var measure: Int := |elems(t)|      // termination
10    unfold tree(t)
11    m := t.val
12    if(t.left != null) {
13      assert |elems(t.left)| < measure  // termination
14      tmp := maxTree(t.left)
15      if(m < tmp) { m := tmp }
16    }
17    if(t.right != null) {
18      tmp := maxTree(t.right)
19      if(m < tmp) { m := tmp }
20    }
21    fold tree(t)
22  }
```

**Figure 7.** A Viper method to compute the maximum in a tree of integers. Permissions to the fields of the tree nodes are represented via the recursive predicate `tree`. The functional behavior is specified in terms of the heap-dependent function `elems`. Both are defined in Fig. 6.

The example in Fig. 7 illustrates these features. It solves the second challenge of the VerifyThis 2011 verification competition. Method `maxTree` computes the maximum in a tree of integers. It takes an instance of the `tree` predicate from its caller and returns it after the call. In order to get access to the fields of the tree node, the method unfolds the predicate in line 10. Before terminating, it re-folds the predicate in line 21.

The functional behavior of `maxTree` is specified in terms of the heap-dependent function `elems`, which is defined in Fig. 6. Heap-dependent functions have a precondition that requires permission to the heap locations accessed by the function. They automatically return all permissions to their caller; a postcondition that provides permissions is neither necessary nor allowed.

Heap-dependent functions are encoded via an uninterpreted function symbol and two axioms. A *definitional axiom* relates the uninterpreted function symbol to the definition of the heap-dependent function. A *framing axiom* expresses that changing heap locations whose permission is not mentioned in the function precondition cannot affect the function value.

The `elems` function traverses the tree recursively and yields the multiset of values stored in the tree nodes. The postcondition of method `maxTree` uses this function to say that (1) the returned value is in the multiset of values stored in the tree, (2) it is no smaller than all other tree elements, and (3) the method does not affect the values stored in the tree. The latter could also be achieved by taking only a fraction of the predicate instance `tree(t)`, allowing callers to hold on to another fraction. The size of the multiset of values is also used as ranking function to prove termination of the recursive method.

## 6. Building Frontend Verifiers

In the previous sections, we have introduced various features of the Viper language and explained how to encode them via translations into simpler intermediate languages down to guarded commands and then to an SMT solver. Since Viper is itself an intermediate verification language, frontend tools use it to encode complex verification problems. In this section, we will discuss a Python version of the tree example form the previous section and its verification in Nagini [11]. The code and specification are shown in Fig. 8.

Nagini requires programs to be statically typed using the mypy type system. It encodes predicate and function definitions as Python methods with annotations `@Predicate` and `@Pure`, resp. Pure methods must be side-effect free. Specifications are expressed as calls to predefined Python methods, which do nothing at run time, but are interpreted by Nagini. This design is adopted from .NET Code Contracts [14] and allows programmers to add annotations without extending the Python syntax.

One such annotation is the `MustTerminate` precondition, which expresses that the method must terminate, and which provides a ranking function that is then encoded via assertions on the Viper level as shown in Fig. 7. Tanslating the Python example from Fig. 8 results in essentially the Viper program from Fig. 7. However, the actual encoding produced by the Nagini verifier is much more complex and includes, for instance, a comprehensive formalization of Python's type system.

## 7. Conclusion

Many modern program verifiers are implemented as a sequence of translations into simpler intermediate verification languages. In these lecture notes, we showed how to encode a range of verification techniques into a simpler guarded-commands language, for which verification condition generation is straightforward. In particular, we showed how to encode access permissions, which allow one to verify heap-manipulating and concurrent programs.

A downside of building verifiers through translations is that error messages for verification failures need to be translated back from the lowest abstraction level to the frontend tool to provide meaningful feedback to programmers. Another drawback is that all translations are part of the trusted codebase, that is, errors in those translations may compromise soundness of the verification. Extracting foundational proofs from translation-based verifiers is an interesting direction for future work.

*Acknowledgement.* We thank Marco Eilers for his help with the Nagini example.

```
from nagini_contracts.contracts import *
from nagini_contracts.obligations import MustTerminate
from typing import Optional, List

class Tree:
  def __init__(self, left: Optional['Tree'], right: Optional['Tree'],
               val: int) -> None:
    self.left = left
    self.right = right
    self.val = val

@Predicate
def tree(self: Tree) -> bool:
  return (Acc(self.left) and Acc(self.right) and Acc(self.val) and
          Implies(self.left is not None, tree(self.left)) and
          Implies(self.right is not None, tree(self.right)))

@Pure
def elems(self: Tree) -> MSet[int]:
  Requires(tree(self))
  Ensures(len(Result()) > 0)
  empty = MSet()  # type: MSet[int]
  return Unfolding(tree(self), MSet(self.val) +
         (elems(self.left) if self.left is not None else empty) +
         (elems(self.right) if self.right is not None else empty))

def maxTree(t: Tree) -> int:
  Requires(tree(t))
  Requires(MustTerminate(len(elems(t))))
  Ensures(tree(t))
  Ensures(0 < elems(t).num(Result()))
  Ensures(Forall(int, lambda e: Implies(0 < elems(t).num(e),
                                        e <= Result())))
  Ensures(elems(t) == Old(elems(t)))
  Unfold(tree(t))
  res = t.val
  if t.left is not None:
    tmp = maxTree(t.left)
    if res < tmp:
      res = tmp

  if t.right is not None:
    tmp = maxTree(t.right)
    if res < tmp:
      res = tmp
  Fold(tree(t))
  return res
```

**Figure 8.** A Nagini version of the tree example from Figs. 6 and 7. Predicate and function definitions are encoded as Python methods with annotations `@Predicate` and `@Pure`, resp. Method specifications and loop invariants are expressed using calls to designated Python methods.

## References

[1] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. Leveraging Rust types for modular specification and verification. Technical report, ETH Zurich, 2018. https://doi.org/10.3929/ethz-b-000311092.

[2] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: The Spec# experience. *Communications of the ACM*, 54(6):81–91, June 2011.

[3] G. Barthe, J. M. Crespo, and C. Kunz. Relational verification using product programs. In M. J. Butler and W. Schulte, editors, *Formal Methods (FM)*, volume 6664 of *LNCS*, pages 200–214. Springer, 2011.

[4] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *Principles of Programming Languages (POPL)*, pages 14–25. ACM, 2004.

[5] S. Blom and M. Huisman. The VerCors tool for verification of concurrent programs. In C. B. Jones, P. Pihlajasaari, and J. Sun, editors, *Formal Methods (FM)*, volume 8442 of *LNCS*, pages 127–131. Springer, 2014.

[6] J. Boyland. Checking interference with fractional permissions. In *Static Analysis Symposium (SAS)*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.

[7] E. M. Clarke, O. Grumberg, D. Kroening, D. Peled, and H. Veith. *Model checking*. MIT Press, 2018.

[8] E. Cohen, M. Moskal, W. Schulte, and S. Tobies. Local verification of global invariants in concurrent programs. In T. Touili, B. Cook, and P. Jackson, editors, *Computer Aided Verification (CAV)*, volume 6174 of *LNCS*, pages 480–494. Springer, 2010.

[9] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL)*, pages 238–252. ACM, 1977.

[10] E. W. Dijkstra. Guarded commands, non-determinancy and a calculus for the derivation of programs. In F. L. Bauer and K. Samelson, editors, *Language Hierarchies and Interfaces*, volume 46 of *LNCS*, pages 111–124. Springer, 1975.

[11] M. Eilers and P. Müller. Nagini: A static verifier for Python. In H. Chockler and G. Weissenbacher, editors, *Computer Aided Verification (CAV)*, volume 10982 of *LNCS*, pages 596–603. Springer, 2018.

[12] M. Eilers, P. Müller, and S. Hitz. Modular product programs. In A. Ahmed, editor, *European Symposium on Programming (ESOP)*, volume 10801 of *LNCS*, pages 502–529. Springer, 2018.

[13] ETH Zurich. *Viper Tutorial*, 2018. viper.ethz.ch/tutorial/.

[14] M. Fähndrich, M. Barnett, and F. Logozzo. Code contracts. http://research.microsoft.com/contracts, 2008.

[15] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *Computer Aided Verification (CAV)*, volume 4590 of *LNCS*, pages 173–177. Springer, 2007.

[16] J.-C. Filliâtre and A. Paskevich. Why3—where programs meet provers. In M. Felleisen and P. Gardner, editors, *European Symposium on Programming (ESOP)*, volume 7792 of *LNCS*, pages 125–128. Springer, 2013.

[17] J. A. Goguen and J. Meseguer. Security policies and security models. In *Security and Privacy*, pages 11–20. IEEE Computer Society, 1982.

[18] S. Heule, I. T. Kassios, P. Müller, and A. J. Summers. Verification condition generation for permission logics with abstract predicates and abstraction functions. In G. Castagna, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 7920 of *LNCS*, pages 451–476. Springer, 2013.

[19] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the CACM*, 12(10):576–580,583, 1969.

[20] A. Lal, S. Qadeer, and S. K. Lahiri. A solver for reachability modulo theories. In P. Madhusudan and S. A. Seshia, editors, *Computer Aided Verification (CAV)*, volume 7358 of *LNCS*, pages 427–443. Springer, 2012.

[21] C. Le Goues, K. R. M. Leino, and M. Moskal. The Boogie verification debugger (tool paper). In G. Barthe, A. Pardo, and G. Schneider, editors, *Software Engineering and*

*Formal Methods (SEFM)*, volume 7041 of *LNCS*, pages 407–414. Springer, 2011.

[22] K. R. M. Leino. This is Boogie 2. `www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/`, 2008.

[23] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.

[24] K. R. M. Leino and P. Müller. Verification of equivalent-results methods. In S. Drossopoulou, editor, *European Symposium on Programming (ESOP)*, volume 4960 of *LNCS*, pages 307–321. Springer, 2008.

[25] K. R. M. Leino and P. Müller. Using the Spec# language, methodology, and tools to write bug-free programs. In P. Müller, editor, *Advanced Lectures on Software Engineering—LASER Summer School 2007/2008*, volume 6029 of *LNCS*, pages 91–139. Springer, 2010.

[26] K. R. M. Leino, P. Müller, and J. Smans. Verification of concurrent programs with Chalice. In A. Aldini, G. Barthe, and R. Gorrieri, editors, *Foundations of Security Analysis and Design V*, volume 5705 of *LNCS*, pages 195–222. Springer, 2009.

[27] L. Moura and N. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[28] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer, 2002.

[29] P. Müller, M. Schwerhoff, and A. J. Summers. Automatic verification of iterated separating conjunctions using symbolic execution. In S. Chaudhuri and A. Farzan, editors, *Computer Aided Verification (CAV)*, volume 9779 of *LNCS*, pages 405–425. Springer, 2016.

[30] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer, 2016.

[31] P. W. O'Hearn. Resources, concurrency and local reasoning. In P. Gardner and N. Yoshida, editors, *Concurrency Theory (CONCUR)*, volume 3170 of *LNCS*, pages 49–67. Springer, 2004.

[32] M. Parkinson and G. Bierman. Separation logic and abstraction. In J. Palsberg and M. Abadi, editors, *Principles of Programming Languages (POPL)*, pages 247–258. ACM, 2005.

[33] J. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS)*, pages 55–74. IEEE Computer Society, 2002.

[34] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[35] M. Schwerhoff. *Advancing Automated, Permission-Based Program Verification Using Symbolic Execution.* PhD thesis, ETH Zurich, 2016.

[36] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 5653 of *LNCS*, pages 148–172. Springer, 2009.

[37] A. J. Summers and P. Müller. Automating deductive verification for weak-memory programs. In D. Beyer and M. Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 10805 of *LNCS*, pages 190–209. Springer, 2018.

[38] V. Vafeiadis and C. Narayan. Relaxed separation logic: a program logic for C11 concurrency. In A. L. Hosking, P. T. Eugster, and C. V. Lopes, editors, *Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 867–884. ACM, 2013.

[39] H. Yang. Relational separation logic. *Theor. Comput. Sci.*, 375(1-3):308–334, 2007.

*Engineering Secure and Dependable Software Systems*
*A. Pretschner et al. (Eds.)*
*IOS Press, 2019*
*© 2019 The authors and IOS Press. All rights reserved.*
*doi:10.3233/978-1-61499-977-5-207*

# Clocks in Kahn Process Networks

Marc Pouzet
*ENS, Département d'informatique, 45 rue d'Ulm, 75230 Paris, France*
Marc.Pouzet@ens.fr

**Abstract.** The language Lustre was introduced to design and implement real-time control software, modeling it as a *continuous function* over treams of datas. A set of equations written in Lustre defines a restricted class of Kahn process networks which can be executed synchronously: all computations can be dated according to a global time scale so that when a value is produced, it is immediately consumed. This restriction is obtained by associating to every stream a *clock* that defines when a value is present or not according to a global time scale. A dedicated type system — the clock calculus — computes a clock for every expression and checks that its actual clock equals its expected clock and thus that intermediate buffers are not needed.

In these course notes,[1] we present a static and dynamic semantics of synchronous Kahn networks. We consider a first-order functional language of streams reminiscent of Lustre and Lucid Synchrone to which we give several denotational semantics. We show that without imposing restrictions, we get two kinds of bad behavior: some networks may deadlock and some cannot execute without unbounded FIFOs. We introduce a clocked semantics and show that the clocking rules correspond to a type system with dependent types. We then extend the language kernel with an explicit `buffer` operator to model communication through a FIFO. The clock calculus is extended with a subtyping rule that is applied where the buffer is used and whose size is inferred. To reduce the complexity of the resolution, we present an abstraction of clocks.

## 1. Introduction

Synchronous languages [3] were introduced about thirty years ago by the concurrent work on three academic languages: Signal [5], Esterel [7] and Lustre [20]. These *domain specific languages* targeted real-time control software, allowing to write modular and mathematically precise system specifications, to simulate, test and verify them, and to automatically translate them into embedded executable code. The environment SCADE,[2] based on a synchronous language [15], is now used routinely to develop various critical control software: in planes (fly-by-wire, engine control, emergency braking), trains (on-board control, interlocking), etc.

All these languages are founded on the synchronous model of time [6] where a system is modeled ideally, with communications and computations assumed to be instantaneous, with formal checks of important safety properties like determinism,

[2]`http://www.esterel-technologies.com/products/scade-suite`

deadlock freedom, execution in bounded time and space, and with a posteriori verification that a given implementation in software or hardware executes quickly enough.

Lustre is a data-flow language: it manipulates infinite streams of data that represent the evolution of an input, an output or a local variable, streams are defined by writing mutually recursive equations over them, and a system is a function from streams to streams. Time is simply the index in a stream. After passing static checks, a stream function is compiled to sequential code (typically C). It can also serve as a functional model of a device or software for the purposes of formal verification ([19] summarises the different uses of Lustre).

A set of stream equations written in Lustre can be interpreted as a *Kahn Process Network* [21]: stream functions are the nodes, every stream defines a communication channel and a set of equations corresponds to a process network. Lustre is Kahnian because a stream function cannot dynamically test whether a signal is present or absent. The consequence is that all execution strategies for a network are guaranteed to compute the same set of streams. Nonetheless, as Lustre targets real-time applications, a function written in Lustre defines a particular subset for which the compiler ensures that it does not deadlock and can be compiled into statically scheduled code running in bounded time and space. This is achieved by imposing a set of static constraints to ensure that the network can be executed synchronously, that is, every computation in the network must be dated according to a global time scale so that when a value is produced, it can be immediately consumed. Hence, no intermediate buffers are needed. This synchronous interpretation is obtained by associating to every stream a *clock* that defines when a value is present or not according to a global time scale. Clocks may or may not be periodic and may depend on input values. A dedicated type system — the clock calculus — computes a clock for every expression and checks that it matches the expected clock.

In this text, we describe a static and dynamic semantics for Lustre from the perspective of Kahn process networks. We consider a simple first-order language of streams reminiscent of Lustre and Lucid Synchrone to which we give several denotational semantics. We show that the naive encoding of streams as lazy data structures gives rise to strange non-causal behaviors, highlighting the need for the prefix order introduced by Kahn. We then give a Kahn semantics to the language kernel. To account for the synchronous restriction, we introduce a clocked semantics and show that the clocking rules that a program must fulfill correspond to typing constraints in a type system with dependent types. We derive a simpler type system which reduces the equality of clocks to name equality. We then extend the language kernel with an explicit buffer operator to model communications via FIFOs. The clock calculus is extended with a subtyping rule that is applied where the buffer is used and whose size is inferred. To reduce the complexity of the resolution, we present an abstraction of clocks.

## *1.1. Kahn Process Networks*

In the 1970s, Kahn studied the semantics of networks of deterministic parallel processes communicating asynchronously through FIFO channels. One may think,

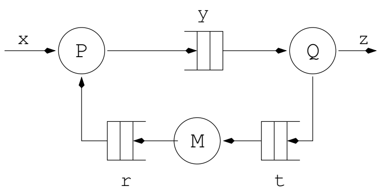


**Figure 1.** A Kahn Process Network with three processes

e.g., of a set of Unix processes communicating via pipes or of threads running asynchronously and synchronising through bounded FIFO queues. Kahn showed that in the case where elementary processes are deterministic, with blocking read on an empty channel and non-blocking writes, the overall network is deterministic — the result does not depend on the relative order in which nodes are activated — and delay insensitive — computation and communication times do not change the network semantics [21,22]. In short, the model is one of the very few that conciliates parallelism and determinism. In a Kahn network, a basic process can be programmed in a sequential language with two primitives: `push` to write a value to a channel and `pop` to read a value. Figure 1 depicts a network with three processes.[3] There is a single reader and writer per channel. A process may only read a single channel at a time and, once committed to reading, it must wait until a value is available. It may not test the channel for emptiness or impose a timeout; that is, it cannot test whether a value is present or absent. One cannot write, for instance:[4]

```
if is_empty a then ...   or   if not (is_empty a) or not (is_empty b)
                                then ...
```

But, it is possible to conditionally read or write according to a value that has been read from a channel, e.g.:

```
let v = pop c in let w = if cond a then pop a else pop b in ...
```

or

```
let v = pop a in if cond v then push a (f v)
```

Figure 2 gives a few examples of elementary primitives: `lift2 f x y z` applies a function `f` pointwise to its two input channels `x` and `y`, and produces an output on channel `z`; the unit delay `fby x y z` concatenates the first element of its input channel `x` to the elements of its second input channel `y` and writes on channel `z`; `merge c x y z` conditionally reads an input channel `x` or `y` according to the value on channel `c` and writes on channel `z`; `split c y z` conditionally writes on channel `y` or `z` according to the value on channel `c`.

Kahn networks with bounded buffers can be implemented by adding a *back pressure* mechanism in order to avoid writes into a full buffer. Nonetheless, this

[3]Figure 15 in Appendix A gives an implementation with threads.
[4]The concrete syntax is that of OCaml.

may introduce artificial blocking if the size of buffers are underestimated. The size of buffers can be increased dynamically [28] but this solution cannot be used for real-time applications where overall memory use must be guaranteed at compile time.

Whether or not a Kahn network is deadlock free or can be executed in bounded memory is undecidable in general [9]. *Synchronous Data Flow* (or SDF) [24] and its variants (*Cyclo Static Data Flow* [27] among others) are restricted classes of networks where every node consumes and produces a fixed number of tokens at every step. The size of buffers can be computed at compile time and a periodic static schedule can be generated. This make SDF suitable for modeling and programming video intensive applications with periodic behavior [32].

To prove that determinism is preserved by composition, Kahn took an approach based on denotational semantics using the following interpretation of channels and processes. A communication channel that carries values of type $T$ is interpreted as a (possibly infinite) sequence of values of type $T$ that describe the *history* of values on the channel. Because a node has its own internal memory, it is interpreted as a function from the histories of its inputs to the histories of its outputs, that is, a stream function. We now recall a few basic properties of sequences, cpos and continuous functions.

### *1.1.1. Sequences and Continuous Functions over Sequences*

Consider a set $T$ of values. $T^n$ denotes the set of sequences of length $n$ made by concatenating elements from $T$. The sequence $v.s$ comprises head $v$ and tail $s$. The empty sequence is written $\epsilon$. The set of finite sequences is written $T^\star = \cup_{n=0}^{\infty} T^n$. The set of finite and infinite sequences of elements of $T$ is written $T^\infty = T^* \cup T^\omega$. We write $\leq$ for the prefix order over sequences; $s \leq s'$ means that $s$ is a prefix of $s'$. For any $s$, $s'$, $\epsilon \leq s$ and if $s \leq s'$, then $v.s \leq v.s'$. A chain in $T^\infty$ is any non-empty subset that is totally ordered by $\leq$. $(T^\infty, \leq, \epsilon)$ is a complete partial order (CPO): $\epsilon$ is its minimum element for the partial order $\leq$ and every chain has a least upper bound. In the case of boolean sequences, where `0` stands for *false* and `1` for *true*, $\epsilon \leq \texttt{0} \leq \texttt{0.1} \leq \texttt{0.1.0} \leq \texttt{0.1.0.0}$ but not $\texttt{1.1} \leq \texttt{1.0}$.

In the sequel, we shall sometimes write a sequence in a more traditional way. A sequence $u = (u_i)_{i \in I}$, finite or not, is a set indexed by an initial segment $I$ of $\mathbb{N}$. $I \subseteq \mathbb{N}$ is an initial segment when $\forall n, m \in \mathbb{N}. (n \in I) \wedge (m \leq n) \Rightarrow (m \in I)$.

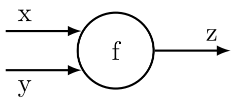
For any subset $A$ of $\mathbb{N}$, there exists a strictly increasing, one-to-one function $\phi_A$ between an initial segment $I_A$ of $\mathbb{N}$ and $A$. An operation that builds a subsequence from a sequence by picking a subset of indices or merges two sequences to build another one corresponds to defining particular $\phi$ functions. This picking does not have to be periodic, as in $(u_{2i})_{i \in \mathbb{N}}$ that is made by taking one element of $u$ every two. It can depend on the value of streams. We shall see concrete examples in the next section.

*General properties of a CPO* If $D_1 = (A_1, \leq_1, \bot_1)$ and $D_2 = (A_2, \leq_2, \bot_2)$ are two cpos, with respective minimum elements $\bot_1$ and $\bot_2$, a function $f : D_1 \to D_2$ is monotonic if and only if for any $x, x' \in D_1$, $x \leq_1 x' \Rightarrow f(x) \leq_2 f(x')$. It is continuous if and only if for any chain $C$ in $D_1$, $f(sup(C)) = sup(\{f(d), d \in C\})$.

```
(* OCaml interface of a FIFO buffer *)
type 'a buff = { push: 'a -> unit; pop: unit -> 'a }
val buffer : unit -> 'a buff
```

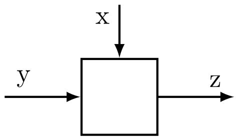
```
(* pointwise application *)
let lift2 f x y z =
  while true do
    let v = x.pop () in
    let w = y.pop () in
    z.push (f v w)
  done
```

$$z = \mathit{lift}_2(f)(x,y)$$



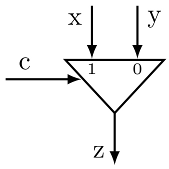
```
(* unit delay *)
let fby x y z =
  let memo = ref (x.pop ()) in
  while true do
    z.push !memo;
    memo := y.pop ()
  done
```

$$z = x\ \texttt{fby}\ y$$



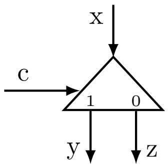
```
(* deterministic merge *)
let merge c x y z =
  while true do
   let v = c.pop () in
   let w = if v then x.pop ()
           else y.pop () in
   z.push w
  end
```

$$z = \texttt{merge}\ c\ x\ y$$



```
(* filter/split a stream *)
let split c x y z =
  while true do
   let cond = c.pop () in
   let v = x.pop () in
   if cond then y.push v
   else z.push v
  done
```

$$y, z = \texttt{split}\ x\ c$$



```
(* sampling *)
let when c x z =
  let y = buffer () in
  split c x z y
```

$$z = x\ \texttt{when}\ c$$

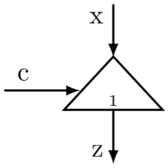


**Figure 2.** A set of data-flow primitives

Any continuous function $f : D \to D$ on a CPO $D = (A, \leq, \bot)$ has a least fix point $\text{fix}\,(f) = lim_{n\to\infty}(f^n(\bot))$, with $f^0(x) = x$ and $f^{n+1}(x) = f(f^n(x))$ (Kleene theorem).

If $A_1$ and $A_2$ are CPOs, then $(D_1 \times D_2, \leq', \bot')$ is also a CPO, with $D_1 \times D_2$

| | | | | | | |
|---|---|---|---|---|---|---|
| $x$ | $x_0$ | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ |
| $y$ | $y_0$ | $y_1$ | $y_2$ | $y_3$ | $y_4$ | $y_5$ |
| $x + y$ | $x_0 + y_0$ | $x_1 + y_1$ | $x_2 + y_2$ | $x_3 + y_3$ | $x_4 + y_4$ | $x_5 + y_5$ |
| $x$ `fby` $y$ | $x_0$ | $y_0$ | $y_1$ | $y_2$ | $y_3$ | $y_4$ |
| | | | | | | |
| $h$ | 1 | 0 | 1 | 0 | 1 | 0 |
| $x' = x$ `when` $h$ | $x_0$ | | $x_2$ | | $x_4$ | |
| $z$ | | $z_0$ | | $z_1$ | | $z_2$ |
| `merge` $h\ x'\ z$ | $x_0$ | $z_0$ | $x_2$ | $z_1$ | $x_4$ | $z_2$ |

**Figure 3.** A set of primitives interpreted as stream functions

being the set of pairs $(x_1, x_2)$ comprising an element $x_1$ from $D_1$ and an element $x_2$ from $D_2$, taking $\bot' = (\bot_1, \bot_2)$ as the minimum element and $\leq'$ such that $(x_1, x_2) \leq' (y_1, y_2) \Leftrightarrow (x_1 \leq_1 y_1) \wedge (x_2 \leq_2 y_2)$. The set $D = (D_1 \mapsto D_2, \leq', \bot')$ where $D_1 \mapsto D_2$ is the set of total continuous functions from $D_1$ to $D_2$, with $f \leq' g \Leftrightarrow \forall s \in D_1. f(s) \leq_2 g(s)$ and $\bot' = (\lambda s. \bot_2)$ is the minimum element, is also a CPO.

### *1.1.2. Application to Kahn Process Networks*

Following the formulation in [21], a network is represented by a set of equations built according to the two following rules:

- If $x_1, ..., x_k$ are the input channels of the network fed with the sequences $i_1, ..., i_k$, add the equations

$$\{x_1 = i_1, \ldots, x_n = i_n\}$$

- Interpret every node $f$ with $n$ input channels $x_1, ..., x_n$ and $p$ output channels $x'_1, ..., x'_p$ as $p$ continuous functions over sequences and add the equations

$$\{y'_1 = f_1(x_1, ..., x_n), \ldots, y'_p = f_p(x_1, ..., x_n)\}$$

The example in Figure 1 is represented by the following set of equations, if $p$ is the stream function associated to process $P$; $\langle q_1, q_2 \rangle$ is associated to $Q$; $m$ to $M$:

$$\{y = p(x, r), z = q_1(y), t = q_2(y), r = m(t)\}$$

Elementary nodes in the network are interpreted as *continuous functions* over sequences. Monotonicity corresponds to the intuition that as a process reads more inputs, it can only produce more outputs: it cannot contradict what has already been produced.

Since every node in a Kahn process network is a continuous function, a set of equations:

| | | |
|---|---|---|
| $d$ ::= | \| `let node` $f$ $pat = e$ | node definition |
| | \| `let clock` $c = ce$ | clock definition |
| | \| $d$ $d$ | sequence of definitions |
| $pat$ ::= | $x$ \| $(pat, ..., pat)$ | pattern |
| $e$ ::= | \| $i$ | constant flow |
| | \| $x$ | flow variable |
| | \| $(e, ..., e)$ | tuple |
| | \| $\texttt{get}_i(e)$ | $i$-th component of a tuple |
| | \| $e$ $op$ $e$ | imported operator |
| | \| `if` $e$ `then` $e$ `else` $e$ | mux operator |
| | \| $f$ $e$ | node application |
| | \| $e$ `where rec` $eqs$ | local definitions |
| | \| $e$ `fby` $e$ | initialized delay |
| | \| $e$ `when` $ce$ \| $e$ `whenot` $ce$ | sampling |
| | \| `merge` $ce$ $e$ $e$ | merging |
| | \| `buffer` $e$ | buffering |
| $eqs$ ::= | $pat = e$ \| $eqs$ `and` $eqs$ | mutually recursive equations |
| $ce$ ::= | $e$ | clock expressions |

**Figure 4.** Language kernel.

$$\{x_1 = f_1(x_1, ..., x_n), ..., x_n = f_n(x_1, ..., x_n)\}$$

has a minimal solution which is $x_1^\infty, ..., x_n^\infty = lim_{j\to\infty}(x_1^j, ..., x_n^j)$ where for all $1 \leq i \leq n$, $x_i^0 = \epsilon$ and $x_i^{j+1} = f_i(x_1^j, ..., x_n^j)$.

The primitives given in Figure 2 [5] can be interpreted as stream functions as illustrated in Figure 3. An important consequence of the interpretation of elementary nodes as continuous functions is that any composition, where some variable may be made local, still defines a continuous function. For the network in Figure 1, if the channels $y$, $r$ and $t$ are considered to be local, the network can be interpreted as a continuous function $f$ of the input $x$, such that the output $z$ satisfies $z = f(x)$.

## 2. A Language of Streams and Stream Functions

We consider a first-order synchronous dataflow language reminiscent of Lustre and Lucid Synchrone but extended with an explicit buffering operator. The syntax is given in Figure 4. A program ($d$) is a sequence of definitions of stream functions called *nodes* and definitions of clock names ($c$). The inputs of a node are described

[5]The operator `when` can also be programmed directly by removing the `else` branch of a `split`. This operator is itself a composition of two `when`.

$$\begin{array}{ll}
op^\sharp(v_1.s_1, v_2.s_2) & = v.op^\sharp(s_1, s_2) \text{ where } v = op(v_1, v_2) \\
\mathtt{fby}^\sharp(v_1.s_1, s_2) & = v_1.s_2 \\
\mathtt{when}^\sharp(v_1.s_1, \mathtt{1}.w) & = v_1.\mathtt{when}^\sharp(s_1, w) \\
\mathtt{when}^\sharp(v_1.s_1, \mathtt{0}.w) & = \mathtt{when}^\sharp(s_1, w) \\
\mathtt{whenot}^\sharp(v_1.s_1, \mathtt{0}.w) & = v_1.\mathtt{whenot}^\sharp(s_1, w) \\
\mathtt{whenot}^\sharp(v_1.s_1, \mathtt{1}.w) & = \mathtt{whenot}^\sharp(s_1, w) \\
\mathtt{merge}^\sharp(\mathtt{1}.w, v_1.s_1, s_2) & = v_1.\mathtt{merge}^\sharp(w, s_1, s_2) \\
\mathtt{merge}^\sharp(\mathtt{0}.w, s_1, v_2.s_2) & = v_2.\mathtt{merge}^\sharp(w, s_1, s_2)
\end{array}$$

**Figure 5.** Then Kahn semantics for the primitives

by a pattern (*pat*) and its body by an expression ($e$). The operators are the basic ones of Lucid Synchrone and their intuitive semantics is detailed later. The expression $e_1$ *op* $e_2$ denotes the pointwise application of a binary operator; `if` $e_1$ `then` $e_2$ `else` $e_3$ is the pointwise application of a conditional; $f\,e$ is the application of a node $f$ to an expression $e$; $e_1$ `fby` $e_2$ conses the head of $e_1$ to $e_2$ and thus corresponds to an initialized delay; $e$ `when` $ce$ samples a stream $e$ according to a boolean expression $ce$ (`whenot` samples when the expression is `0`). We call this boolean expression a clock. The operator `merge` $ce$ $e_1$ $e_2$ merges two streams according to a clock. Finally `buffer` $e$ buffers $e$. We write $e$ `where rec` *eqs* for an expression defined by a collection of mutually recursive equations (*eqs*). The basic data-flow primitives of this language kernel are those of Figure 2. For the language of clocks $ce$, we take any boolean expression. We shall later consider particular cases of this language.

### *2.1. Denotational Semantics*

We first give a denotational semantics based on possibly infinite sequences, following the interpretation given by Kahn. In this setting, the operator `buffer` is simply the identity function. We then define a synchronous semantics which characterises the evolution of the streams and the contents of the buffers.

*Notation for the semantics.* We write $\rho$ for an environment and $\rho(x)$ for the value associated to the variable $x$ in the environment $\rho$. The environment $\rho+[x \leftarrow v]$ is the environment $\rho$ to which has been added the binding of $x$ to $v$. The environment $\rho+\rho'$ is the environment that contains the associations of the environment $\rho$ and the associations of the environment $\rho'$ provided that no single variable appears in both $\rho$ and $\rho'$.

The interpretation of an expression $e$ in an environment $\rho$ is written $[\![e]\!]_\rho$. This notation will also be used for the denotation of equations and declarations.

Finally, when presenting the interpretation of the primitives as stream functions, we shall use the notation $\sharp$ as an exponent to distinguish syntactic constructs from their interpretations.

### 2.1.1. A Kahn semantics

Every node declaration `let node` $f\ pat = e$ is interpreted as a continuous function over sequences. The Kahn semantics for the primitives of the language is given in Figure 5.

- *op* applies an imported operator pointwise on scalar values;
- `fby` is the unit delay; it appends the head of its first argument onto the value of its second argument;[6]
- `when` is the sampling operator: it passes its input to its output only if the condition is true (value `1`) and otherwise does not produce any output;
- `whenot` is the complementary sampling operator which passes its input to its output only if the second input is false (value `0`) and otherwise does not produce any output;
- `merge` merges two input streams according to a boolean condition. It passes its first input to its output when the boolean condition is true and the second input otherwise.

These definitions must be completed to deal with the empty sequence $\epsilon$. All operators return $\epsilon$ if one of their arguments is the empty sequence ($\epsilon$ is absorbing), except for the operator `fby` which is such that $\texttt{fby}^\sharp(v.s, \epsilon) = v.\epsilon$. We also have to deal with possible type errors. Several solutions can be taken: (1) complete all the definitions by returning $\epsilon$ in case of a type error; (2) add a special `TypeError` value to the set of streams and transmit this value; (3) define the semantics for well-typed expressions only. For the sake of simplicity, we apply the first solution.

All the primitives are monotonic and continuous [10].

The semantics of expressions of the language is defined in Figure 6. The definition uses the interpretations of the primitives given previously. We write $[\![e]\!]_\rho$ to denote the value of $e$ in the environment $\rho$. We ensure that the language is first order by using two distinct namespaces: one that maps local variables to (stream) values or tuples of values, and a second that maps global variables to functions. The environment $\rho$ is thus a pair $(\rho_s, \rho_n)$ where $\rho_s$ associates a value to every free variable of $e$ and $\rho_n$ associates a value to every function. Letting $\mathit{Var}_s$ denote the set of variable names and $\mathit{Var}_n$ the set of node names, we have

$$\begin{array}{lll} \mathit{Stream}(T) & = T^\infty & \text{sequences} \\ V & = \mathit{Stream}(T_1) + \cdots + \mathit{Stream}(T_n) + V \times \cdots \times V & \text{values for local variables} \\ \rho_s & :\ \mathit{Var}_s \rightarrow V & \text{local environment} \\ \rho_n & :\ \mathit{Var}_n \rightarrow (V \rightarrow V) & \text{global environment} \end{array}$$

If $\rho = (\rho_s, \rho_n)$ and $\rho' = (\rho'_s, \rho'_n)$ then $\rho + \rho' = (\rho_s + \rho'_s, \rho_n + \rho'_n)$. We write $\rho + [z \leftarrow v]$ to add the association $z \leftarrow v$ in the appropriate part of the pair $\rho$.

The interpretation of $e$ `where rec` *eqs* uses the interpretation of the set of equations *eqs* as the supplementary environment. If *eqs* is $x_1{=}e_1$ `and` $\cdots$ `and` $x_k{=}e_k$, its interpretation is an environment that associates every variable $x_i$ with the interpretation of $e_i$:

[6]It corresponds to the `A` operator of [21]. The `fby` operator was introduced in Lucid [2] and used in [10].

$$\begin{array}{ll}
[\![i]\!]_\rho & = i.[\![i]\!]_\rho \\
[\![x]\!]_\rho & = \rho_s(x) \\
[\![\texttt{(}e_1\texttt{,}...\texttt{,}e_n\texttt{)}]\!]_\rho & = ([\![e_1]\!]_\rho, ..., [\![e_n]\!]_\rho) \\
[\![\texttt{get}_i(e)]\!]_\rho & = s_i \text{ if } [\![e]\!]_\rho = (s_1, ..., s_n) \\
[\![e_1 \; op \; e_2]\!]_\rho & = op^\sharp([\![e_1]\!]_\rho, [\![e_2]\!]_\rho) \\
[\![f \, e]\!]_\rho & = \rho_n(f) \, [\![e]\!]_\rho \\
[\![e \texttt{ where rec } eqs]\!]_\rho & = [\![e]\!]_{\rho+\rho'} \text{ where } \rho' = ([\![eqs]\!]_\rho, \emptyset) \\
[\![e_1 \texttt{ fby } e_2]\!]_\rho & = \texttt{fby}^\sharp([\![e_1]\!]_\rho, [\![e_2]\!]_\rho) \\
[\![e \texttt{ when } ce]\!]_\rho & = \texttt{when}^\sharp([\![e]\!]_\rho, [\![ce]\!]^{\mathsf{ce}}_\rho) \\
[\![e \texttt{ whenot } ce]\!]_\rho & = \texttt{whenot}^\sharp([\![e]\!]_\rho, [\![ce]\!]^{\mathsf{ce}}_\rho) \\
[\![\texttt{merge } ce \; e_1 \; e_2]\!]_\rho & = \texttt{merge}^\sharp([\![ce]\!]^{\mathsf{ce}}_\rho, [\![e_1]\!]_\rho, [\![e_2]\!]_\rho) \\
[\![e]\!]^{\mathsf{ce}}_\rho & = [\![e]\!]_\rho \\
[\![\texttt{buffer } e]\!]_\rho & = [\![e]\!]_\rho
\end{array}$$

**Figure 6.** The Kahn semantics for the language expressions

$$[\![x_1 = e_1 \texttt{ and } \cdots \texttt{ and } x_k = e_k]\!]_\rho = [x_1 \leftarrow {x_1}^\sharp, \ldots, x_k \leftarrow {x_k}^\sharp]$$
$$\text{where } {x_1}^\sharp, \ldots, {x_k}^\sharp = \text{fix}\left(\lambda s_1, \ldots, s_k. \, [\![e_1]\!]_{\rho+[x_1 \leftarrow s_1, \ldots, x_k \leftarrow s_k]}, \ldots, [\![e_k]\!]_{\rho+[x_1 \leftarrow s_1, \ldots, x_k \leftarrow s_k]}\right)$$

The interpretation of the operators `when`, `whenot` and `merge` uses the interpretation of their clock argument $ce$. In this basic language, we consider that a clock expression can be any boolean expression, hence $[\![ce]\!]_\rho = [\![e]\!]_\rho$. In the second part of these notes, we introduce a dedicated sublanguage of boolean expressions.

The operation `buffer` copies its input into its output, possibly delaying it. Since the Kahn semantics is unable to express timing, the interpretation here is simply the identity function.

The semantics of a program is defined as follows:

$$\begin{array}{ll}
[\![\texttt{let node } f \; x = e]\!]_\rho = \rho + [f \leftarrow (\lambda s. \, [\![e]\!]_{\rho+[x \leftarrow s]})] & \\
[\![\texttt{let clock } c = ce]\!]_\rho = \rho + [c \leftarrow [\![ce]\!]^{\mathsf{ce}}_\rho] & \\
[\![d_1 \; d_2]\!]_\rho \qquad = [\![d_2]\!]_{\rho+\rho_1} \text{ where } \rho_1 = [\![d_1]\!]_\rho &
\end{array}$$

The evaluation of a program $d$ having $f$ as the main node in an environment where the input stream is $I$ is defined by:

$$\rho_n(f) \; I \text{ where } (\rho_s, \rho_n) = [\![d]\!]_{(\emptyset,\emptyset)}$$

*2.1.2. Reformulating the Kahn semantics using indexed streams*

The semantics can also be formulated using the notation $(u_i)_{i \in N}$ with $N \subseteq \mathbb{N}$ being an initial section of $\mathbb{N}$. If $A \subseteq \mathbb{N}$, there exists a one-to-one function $\phi_A$ between an initial section $I_A$ and $A$.

$$\begin{array}{lll}
\mathit{lift_0}^\sharp(v) & = (u)_{n \in \mathbb{N}} & \mathit{with}\ \forall n \in \mathbb{N}.\, u_n = v \\
\mathit{lift_1}^\sharp(\mathit{op})((u_n)_{n \in N}) & = (v_n)_{n \in N} & \mathit{with}\ \forall n \in N.\, v_n = \mathit{op}(u_n) \\
\mathit{lift_2}^\sharp(\mathit{op})((u_n)_{n \in N}, (v_n)_{n \in N}) & = (w_n)_{n \in N} & \mathit{with}\ \forall n \in N.\, w_n = \mathit{op}(u_n, v_n) \\
\texttt{fby}^\sharp((u_n)_{n \in N}, (v_n)_{n \in N}) & = (w_n)_{n \in N} & \mathit{with}\ w_0 = u_0 \\
 & & \mathit{and}\ \forall n \in N \backslash \{0\}.\, w_n = v_{n-1}
\end{array}$$

If $(h_n)_{n \in N}$ is a boolean sequence, define $N_h$ and $N_{\overline{h}}$ as a partition of $N$:

$$N_h = \{k \in N \mid h_k = \mathtt{1}\} \quad \text{and} \quad N_{\overline{h}} = \{k \in N \mid h_k = \mathtt{0}\}$$

The filter operator `when` and complement `merge` are defined in the following way:

$$\begin{array}{lll}
\texttt{when}^\sharp((u_n)_{n \in N}, (h_n)_{n \in N}) & = (v_n)_{n \in I_{N_h}} & \mathit{with}\ v_n = u_{\phi_{N_h}(n)} \\
\texttt{merge}^\sharp((h_n)_{n \in N}, (u_n)_{n \in I_{N_h}}, (v_n)_{n \in I_{N_{\overline{h}}}}) & = (w_n)_{n \in N} & \mathit{with}\ w_n = u_n\ \mathit{if}\ n \in N_h \\
 & & \mathit{and}\ w_n = v_n\ \mathit{if}\ n \in N_{\overline{h}}
\end{array}$$

A set of equations over sequences becomes a set of mutually recursive functions, from natural numbers to values. Figure 7 gives a possible implementation in OCaml.

We use the functions `index` and `cumul` which are, respectively, the index and cumulative functions, written $I$ and $O$ in [25]. If $h$ is a boolean stream, $O(h)(n)$ is the sum of 1s up to index $n$; $I(h)(n)$ is the index of the $n$th 1 in $h$.

$$O(h)(n) = \sum_{i=0}^{n} h(i) \quad I(h)(n) = \min\left(\{k \in \mathbb{N} \mid O_h(k) = n\}\right)$$

This implementation, however, only addresses sequences that are total over $\mathbb{N}$. Trying to compute `index(x when (constant false))` for any $n$ results in a stack overflow. This is no surprise since the domain of `when` can be finite. We shall see later how this problem can be addressed.

Moreover, even in the case where all sequences are infinite, the implementation is extremely inefficient. While it is useful for reasoning about programs, it is not a practical implementation. Indeed, the value of a sequence $x$ at instant $n$ is computed recursively, possibly back to index 0, with no reuse of previously computed values. It is possible, though, to program the initial Kahn semantics almost directly using infinite data structures and lazy evaluation.

```
(* sequences as functions *)
let lift0 v n = v
let constant = lift0

let lift1 op x n = op (x(n))
let lift2 op x y n = op (x(n)) (y(n))
let lift3 op x y z n = op (x(n)) (y(n)) (z(n))

let notl x = lift1 not

let fby x y = if n = 0 then x 0 else y(n-1)

let when x h n = x(index(h)(n+1))
let merge h x y n = if h(n) then x(cumul(h)(n)) else y(cumul(notl h)(n)

(* cumulative and index functions *)
let rec cumul(h)(n) = h(n) + (if n = 0 then 0 else cumul(h)(n-1))

let rec index(h)(n) = index_aux(h)(0)(n)
and index_aux(h)(i)(n) =
      if h(i) then if n = 1 then i else index_aux(h)(i+1)(n-1)
      else index_aux(h)(i+1)(n)
```

**Figure 7.** An implementation of sequences as functions in OCaml

```
module Streams where

-- lifting constants
constant x = x : (constant x)

-- pointwise application
extend (f:fs) (x:xs) = (f x):(extend fs xs)

lift1 op xs = extend (constant op) xs
lift2 op xs ys = extend (extend (constant op) xs) ys
lift3 op xs ys zs = extend (extend (extend (constant op) xs) ys) zs

-- delays
(x:xs) `fby` y = x:y
pre x y = x : y

-- sampling
(x : xs) `when` (True : cs)  = (x : (xs `when` cs))
(x : xs) `when` (False : cs) = xs `when` cs

merge (True : c) (x : xs) y  = x : (merge c xs y)
merge (False : c) x (y : ys) = y : (merge c x ys)
```

**Figure 8.** A Haskell implementation with (lazy) lists

### *2.1.3. An implementation in Haskell with lazy lists*

The definitions in Figure 6 can be implemented with potentially infinite data structures and lazy evaluation. Figure 8 gives an implementation in Haskell using lists which can then be used to write many stream functions. For example,

```
plusl x y = lift2 (+) x y
minusl x y = lift2 (-) x y

-- integers greaters than n
from n =
  let nat = n `fby` (plusl nat (constant 1)) in nat

-- a resettable counter
reset_counter res input =
  let output = ifthenelse res (constant 0) v
      v = ifthenelse input
                     (pre 0 (plusl output (constant 1)))
                     (pre 0 output)
  in output

-- a periodic clock
every n =
  let o = reset_counter (pre 0 o = plusl n (constant 1)) (constant True)
  in o

filter n top = top `when` (every n)

hour_minute_second top =
  let second = filter (constant 10) top in
  let minute = filter (constant 60) second in
  let hour = filter (constant 60) minute in
  hour, minute, second
```

Yet, we have essentially just written Lustre functions that pass the compilation checks. The two following functions cannot be written in Lustre. The first one computes the sequence $(o_n)_{n \in \mathbb{N}}$ from an input $(x_n)_{n \in \mathbb{N}}$ such that $o_{2n} = x_n$ and $o_{2n+1} = x_n$.

```
-- the half clock
half = (constant True) `fby` notl half

-- double its input
stutter x =
  o = merge half x ((pre 0 o) when notl half) in o
```

This is an example of an oversampling function: its internal rate is faster than the rate of its input. This program can be implemented in bounded memory and time. But the Lustre compiler rejects oversampling functions. Another example of an oversampling function is one that computes the root of an input $x$ using the Newton method.[7] It mimics an internal while loop.

[7]This example is due to Paul Le Guernic and was originally written in Signal.

$$u_n = u_{n-1}/2 + x/2u_{n-1} \qquad u_1 = x$$

```
eps = constant 0.001
`div` = lift2 (\x y -> x div y)
`minus` = lift2 (\x y -> x - y)
`lessthan` = (lift2 (\x y -> x <= y)

root input =
  let ic = merge ok input ((pre 0.0 ic) `when` (notl ok))
      uc = ((pre 0.0 uc) `div` (constant 2.0)) `plusl`
            (ic `div` ((constant 2.0) `times` (pre 0.0 uc)))
      ok = (constant true) `->`
              ((uc `minus` (pre 0.0 uc)) `lessthan` eps
      output = uc `when` ok
  in output
```

Of course, there are many other valid programs that cannot be written in Lustre, in particular those that exploit the expressiveness of Haskell and its type system, like the possibility to write higher order functions.

**Remark 2.1** (Finite values encoded as infinite ones)**.** A classic way to represent $\epsilon$ with a coinductive type that represents only infinite values, is to infinitely repeat a distinguished value. This approach is used, for example, in [8] and [29]. Instead of interpreting streams as lists, we can instead define streams as:

```
data Value a = Value a | Eps
data Stream a = Cons a (Stream a)
data StreamEps a = Stream (Value a)

eps = Cons Eps eps
one = Cons 1 eps
```

We shall see, however, that the encodings as lazy data-structures we have considered model neither synchrony nor causality.

### *2.1.4. Where are the monsters?*

*Causality monsters* In the above encoding, a stream is represented as a lazy data structure. Laziness, however, allows streams to be built in a strange manner. The following definitions are perfectly valid and produce infinite streams for `one`, `x` and `output`.

```
hd (x:y) = x
tl (x:y) = y
incr (x:y) = (x+1) : incr y

one = 1 : one
x = (if hd(tl(tl(tl(x)))) = 5 then 3 else 4) : 1 : 2 : 3 : one
output = (hd(tl(tl(tl(x))))) : (hd(tl(tl(x)))) : (hd(x)) : x
```

The values are `x` $= 4 : 1 : 2 : 3 : 1 : \cdots$ and `output` $= 3 : 2 : 4 : 3 : 2 : 4 : \cdots$. Streams are implemented as an inductive data structure, `x` and `output` are computed sequentially:

- $x^0 = \perp$, $x^1 = \perp : 1 : 2 : 3 : one$, $x^2 = 4 : 1 : 2 : 3 : one$.
- $output^0 = \perp$, $output^1 = 3 : 2 : 4 : \cdots$

Another example:[8]

```
next x = tl x
nat = zero `fby` (incr nat)
ifn n x y = if n <= 9 then hd(x) : if9 (n+1) (tl(x)) (tl(y)) else y
if9 x y = ifn 9 x y

x = if9 (incr (next x)) nat
```

The output stream is $x = 18 : 17 : 16 : 15 : 14 : 13 : 12 : 11 : 10 : 9 : 10 : 11 : \cdots$.

Are these reasonable programs? Streams are constructed in a reverse manner from the future to the past. They do not obey the intuition that we have of causality, that streams must be constructed from left to right. This is because the structural order between streams allows to fill in the holes in any order, e.g.:

$$(\perp : \perp) \leq (\perp : \perp : \perp : \perp) \leq (\perp : \perp : 2 : \perp) \leq (\perp : 1 : 2 : \perp) \leq (0 : 1 : 2 : \perp)$$

Note that it is possible to build streams with intermediate holes, that is, with undefined values in the middle, from which one can build other streams without holes:

$$half = 0 : \perp : 0 : \perp : \cdots$$

```
fail = fail
half = 0:fail:half
fill x = (hd(x)) : fill (tl(tl x))
ok = fill half
```

The definition of streams in Figure 8 follows the structural order between data structures, which is also the order between functions: $\perp \leq_{\mathsf{struct}} v$ and the structure $(v : w)$ is less defined than $(v' : w')$ when $v$ is less defined than $v'$ and $w$ is less defined than $w'$: $(v : w) \leq_{\mathsf{struct}} (v' : w') \Leftrightarrow v \leq_{\mathsf{struct}} v' \wedge w \leq_{\mathsf{struct}} w'$. It does not model the intuition of causality that values in the stream must be computed from left to right. The prefix order is thus preferable, that is, $\perp \leq x$ and $v : x \leq v : y \Leftrightarrow x \leq y$.

[8]This example is due to Paul Caspi [4]

```
module SStreams where
-- only consider streams where the head is always a value (not bot)
data ST a = Cons !a (ST a) deriving Show
constant x = Cons x (constant x)

extend (Cons f fs) (Cons x xs) = Cons (f x) (extend fs xs)

(Cons x xs) `fby` y = Cons x y

(Cons x xs) `when` (Cons True cs)  = (Cons x (xs `when` cs))
(Cons x xs) `when` (Cons False cs) = xs `when` cs

merge (Cons True c)  (Cons x xs) y  = Cons x (merge c xs y)
merge (Cons False c) x (Cons y ys)  = Cons y (merge c x ys)
```

**Figure 9.** A Haskell implementation with (lazy) lists that enforces causality

*Remark:* This order can be adapted to functions from natural numbers to values, allowing to have intermediate holes in results [4].

$$(x \leq' y) \Leftrightarrow (\forall n \in \mathbb{N}.\, x(n) \leq y(n) \Rightarrow \forall m \geq n.\, x(m) = \bot)$$

For example, the following sequence is ordered:

$$\bot \leq (1.\bot) \leq (1.\bot.2.\bot) \leq (1.\bot.2.\bot.3.\bot)$$

Under the prefix ordering, all the previous strange programs denote $\bot$.

*Causal function:* A function from streams to streams, is said to be *causal* when it is monotonic for the prefix order. This definition may seem too permissive as the function `next` (or `tl`), given below and presented like the following is considered to be causal.

$$\forall n \in \mathbb{N}.\, \mathtt{next}(x)(n) = x(n+1)$$

Indeed, the operator `next` can be programmed and is perfectly valid (up to syntactic details) in Lucid Synchrone (and also Lucy-n), for example.

```
let node next x = x when (false fby true)
```

We shall see in the next section how the use of such functions must nonetheless be constrained.

Possibly non-causal streams can be proscribed by forbidding values of the form $\bot.x$. Figure 9 gives a simple modification of the previous definitions in Haskell. The annotation `!a` forces the first argument of the stream constructor `Cons` to be strict, that is, to evaluate to a value. Now all the previous strange, non-causal programs have value $\bot$.

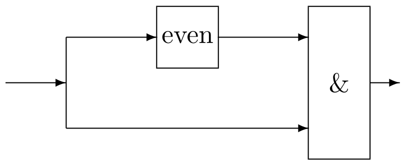


**Figure 10.** A non synchronous example

*Some "synchrony" monsters* Another kind of strange behavior can occur. Consider the input sequence $x = (x_i)_{i\in I\!N}$ and the function `even` such that $\texttt{even}(x) = (x_{2i})_{i\in I\!N}$. Define the equation $y = x \,\&\, \texttt{even}(x)$. It should define the sequence $(x_i \,\&\, x_{2i})_{i\in I\!N}$. In Haskell, given the definitions of Figure 9, we have:

```
even (Cons x (Cons y xs)) = Cons x (even xs)
and_gate (Cons x xs) (Cons y ys) = Cons (x && y) (and xs ys)
```

Figure 10 depicts the corresponding Kahn network. The fork on the left implicitly represents a simple duplication operator. Even though the `even` and & blocks are finite-memory processes, the composition cannot be executed in bounded memory. As time goes by, the size of the FIFO of the bottom line increases and must eventually overflow.

In real-time applications, sucha compositions must be statically rejected. Moreover, all the synchronization is hidden in communication channels. Finally, even in the case where the overall memory can be statically bounded, our Haskell encoding needs a complicated runtime system, with allocation and deallocation of intermediate stream values at every step and a garbage collector. There are no real surprises here. The Kahn semantics models neither time nor the resources necessary to synchronise values. If bounded FIFOs are explicitly managed, their size has to be determined, and this can lead to possible deadlocks.

### *2.1.5. Clocked Streams*

To account for precise synchronisations between nodes, we introduce a new semantics in which the use of data-flow primitives is restricted. We shall consider that all streams progress synchronously, each producing at global steps either a standard value or the special explicit value *abs* denoting that a value is *absent*, that is, not yet present. The size and content of buffers is also made explicit.

*AbsStreamT* defines the set of clocked sequences made of values from the set $T_{abs} = T + \{abs\}$.

$$\begin{aligned} T_{abs} &= T + \{abs\} \\ AbsStream(T) &= (T_{abs})^{\infty} \end{aligned}$$

It is a sequence of present and absent values that can be represented in Haskell as follows.

```
data maybe a = Present a | Absent
data AbsStream a = ST (maybe a)
```

The clock of a sequence $s$ is a boolean sequence that indicates when a value is present. For that, we define the function *clock* between clocked sequences and boolean sequences:

$$\begin{aligned}
\texttt{bool} &= \{\texttt{0}, \texttt{1}\} \\
Clock &= \texttt{bool}^{\infty} \\[4pt]
clock(\epsilon) &= \epsilon \\
clock(abs.x) &= \texttt{0}.clock(x) \\
clock(v.x) &= \texttt{1}.clock(x)
\end{aligned}$$

We now make the link between the clock and the set of present/absent values more precise by defining:

$$ClockedStream(T)(c) = \{s \mid s \in (T^{abs})^{\infty} \wedge clock(s) \leq c\}$$

For a boolean sequence $c$, $ClockedStream(T)(c)$ is the set of sequences with clock $c$. It is prefix closed: if $s$ is a prefix of $s'$ with clock $c$, that is, $s' \in ClockedStream(T)(c)$, $s \in ClockedStream(T)(c)$.

The *synchronous semantics* is defined by reinterpreting the basic primitives over clocked sequences. We can replay the Kahn semantics in Section 2.1.1. It is defined by $[\![e]\!]_{\rho}^{\mathsf{abs}}$ which computes the value of $e$ in an environment $\rho = (\rho_s, \rho_n)$. The set of values is replaced by:

$$V = AbsStream(T_1) + \cdots + AbsStream(T_n) + V \times \cdots \times V \text{ values for local variables}$$

The semantics of expressions, equations and global definitions is essentially unchanged. What changes is the interpretation of primitives on which we concentrate now.

In the following, we write $\epsilon$ for the empty sequence; $v$ for a present value and *abs* for an absent value. Hence, $v.s$ denotes a clocked sequence whose head is present and $abs.s$ denotes a sequence whose head is absent.

The interpretation over clocked sequences for the primitives of the language is summarised in Figure 11. We start with the simplest operators, the generator of a constant sequence from a scalar value and the operator to lift a scalar function pointwise over input sequences.

To give a clocked semantics for the constant generator, we need an extra argument to determine whether the current value is present or not, that is:

$$\begin{aligned}
\texttt{const}^{\sharp}(i, \texttt{1}.w) &= i.\texttt{const}^{\sharp}(i, w) \\
\texttt{const}^{\sharp}(i, \texttt{0}.w) &= abs.\texttt{const}^{\sharp}(i, w)
\end{aligned}$$

Thus, $\texttt{const}^{\sharp}(i, w)$ defines a constant stream with clock $w$, that is, $clock(\texttt{const}^{\sharp}(i, w)) = w$.

Consider now the semantics of $s + s'$, for example. At least two situations can occur. If the two inputs are absent, we propagate the absent on the output. If the two inputs are present, we output the sum of the two.

$$\begin{array}{ll}
\texttt{const}^\sharp(i, 1.w) & = i.\texttt{const}^\sharp(i, w) \\
\texttt{const}^\sharp(i, 0.w) & = abs.\texttt{const}^\sharp(i, w) \\[1em]
op^\sharp(abs.s_1, abs.s_2) & = abs.op^\sharp(s_1, s_2) \\
op^\sharp(v_1.s_1, v_2.s_2) & = v.op^\sharp(s_1, s_2) \text{ where } v = op(v_1, v_2) \\[1em]
\texttt{fby}^\sharp(abs.s_1, abs.s_2) & = abs.\texttt{fby}^\sharp(s_1, s_2) \\
\texttt{fby}^\sharp(v_1.s_1, v_2.s_2) & = v_1.\texttt{fby1}^\sharp(v2, s_1, s_2) \\
\texttt{fby1}^\sharp(v, abs.s_1, abs.s_2) & = abs.\texttt{fby1}^\sharp(v, s_1, s_2) \\
\texttt{fby1}^\sharp(v, v_1.s_1, v_2.s_2) & = v.\texttt{fby1}^\sharp(v_2, s_1, s_2) \\[1em]
\texttt{when}^\sharp(abs.s_1, abs.w) & = abs.\texttt{when}^\sharp(s_1, w) \\
\texttt{when}^\sharp(v_1.s_1, 1.w) & = v_1.\texttt{when}^\sharp(s_1, w) \\
\texttt{when}^\sharp(v_1.s_1, 0.w) & = abs.\texttt{when}^\sharp(s_1, w) \\
\texttt{whenot}^\sharp(abs.s_1, abs.w) & = abs.\texttt{whenot}^\sharp(s_1, w) \\
\texttt{whenot}^\sharp(v_1.s_1, 0.w) & = v_1.\texttt{whenot}^\sharp(s_1, w) \\
\texttt{whenot}^\sharp(v_1.s_1, 1.w) & = abs.\texttt{whenot}^\sharp(s_1, w) \\[1em]
\texttt{merge}^\sharp(abs.w, abs.s_1, abs.s_2) & = abs.\texttt{merge}^\sharp(w, s_1, s_2) \\
\texttt{merge}^\sharp(1.w, v_1.s_1, abs.s_2) & = v_1.\texttt{merge}^\sharp(w, s_1, s_2) \\
\texttt{merge}^\sharp(0.w, abs.s_1, v_2.s_2) & = v_2.\texttt{merge}^\sharp(w, s_1, s_2) \\[1em]
\texttt{buffer}^\sharp(v.s, n, abs.s_1, 1.w) & = v.\texttt{buffer}^\sharp(s, n+1, s_1, w) \\
\texttt{buffer}^\sharp(v.s, n, v_1.s_1, 1.w) & = v.\texttt{buffer}^\sharp(s.v_1, n, s_1, w) \\
\texttt{buffer}^\sharp(\epsilon, n, v_1.s_1, 1.w) & = v_1.\texttt{buffer}^\sharp(\epsilon, n, s_1, w) \\
\texttt{buffer}^\sharp(s, n, abs.s_1, 0.w) & = abs.\texttt{buffer}^\sharp(s, n, s_1, w) \\
\texttt{buffer}^\sharp(s, n, v_1.s_1, 0.w) & = abs.\texttt{buffer}^\sharp(s.v_1, n-1, s_1, w) \text{ if } n > 0
\end{array}$$

**Figure 11.** The clocked semantics for the primitives

$$\begin{array}{ll}
op^\sharp(abs.s_1, abs.s_2) & = abs.op^\sharp(s_1, s_2) \\
op^\sharp(v_1.s_1, v_2.s_2) & = v.op^\sharp(s_1, s_2) \text{ where } v = op(v_1, v_2)
\end{array}$$

It is tempting to add:

$$\begin{array}{l}
op^\sharp(abs.s_1, v_2.s_2) = abs.op^\sharp(s_1, v_2.s_2) \\
op^\sharp(v_1.s_1, abs.s_2) = abs.op^\sharp(v_1.s_1, s_2)
\end{array}$$

to complete with the default rule for absent values as in the initial Kahn semantics. A benefit of having added an absent value to the set of instantaneous values is that we no longer need to deal with both finite and infinite sequences. The empty sequence is simply represented as the infinite sequence $abs^\omega$ and finite sequences are simply completed to infinite ones by suffixing them with $abs^\omega$. The synchronous monsters, however, have not been eradicated!

The synchronous aspect comes from the absence of certain definitions. For example, there is no definition to evaluate $op^\sharp(v_1.s_1, abs.s_2)$ nor $op^\sharp(abs.s_1, v_2.s_2)$, that is, both inputs must be simulataneously present or absent. Otherwise, one of them should be buffered.

### *2.1.6. Dealing with partial definitions: the clock calculus*

What happens when one element is present and the other is absent? One idea is to statically reject these cases by requiring + to have the following type:

$$(+) : \forall cl : Clock.\ ClockedStream(\texttt{int})(cl) \times ClockedStream(\texttt{int})(cl) \rightarrow ClockedStream(\texttt{int})(cl)$$

In words, (+) expects its input streams to be on the same clock *cl* and guarantees to produce its output on that clock. These conditions are expressed in the form of a type that must be verified statically. This idea is exploited in [8] by defining clocked sequences in Coq as a coinductive dependent type: the type constraint for (+) and other operators, and the clock constraints for expressions, equations and functions are performed directly by the Coq type checker.

**Remark 2.2** (Two type systems versus a single one)**.** There has been long debate about whether the so-called clock calculus for Lustre, Scade 6, Lucid Synchrone and Signal should merge both classical type information about data and presence/absence information. For Lustre and Signal, the clock calculus was not expressed as a type system and was applied after (classical) static typing. Separating the two, we have two signatures for (+), computed by two different type systems:

$$\begin{array}{ll} (+) : \texttt{int} \times \texttt{int} \rightarrow \texttt{int} & \text{type signature} \\ (+) : \forall cl.\ cl \times cl \rightarrow cl & \text{clock signature} \end{array}$$

For Lucid Synchrone,[9] we also decided to separate the type system for data from that for clocks; the compiler thus calculates the two types given above. One of the reasons is that the compiler implements two other type systems, one that ensures the absence of instantaneous loops and another that analyses uses of the uninitialised delay `pre`. After much trial and error, we found it simpler to implement the various systems separately. Moreover, typing occurs sequentially (datatypes, clocks, causality, initialization) so that the information produced by earlier passes is reused by later ones. In particular, the skeleton for types is used to simplify the inference of clock types, causality types and initialization types.

Nonetheless, having several type systems adds useless redundancy in the implementation. It also complicates the formulation of correctness properties. Each of the systems precludes a particular kind of error. It also adds redundancy in interfaces, for example, if one wants to declare a data structure or function that requires a specific clock type. The debate is unfinished. In his thesis [17] and paper [18], Guatto follows an alternative approach that mixes regular type information and clock information, where the clocks express a form of modality in the spirit of guarded types.

In the following, we only consider clock types. Let us consider the case of the unit delay `fby`.

[9]`https://www.di.ens.fr/~pouzet/lucid-synchrone/`

$$\begin{array}{ll}
\texttt{fby}^\sharp(abs.s_1, abs.s_2) & = abs.\texttt{fby}^\sharp(s_1, s_2) \\
\texttt{fby}^\sharp(v_1.s_1, v_2.s_2) & = v_1.\texttt{fby1}^\sharp(v_2, s_1, s_2) \\
\texttt{fby1}^\sharp(v, abs.s_1, abs.s_2) & = abs.\texttt{fby1}^\sharp(v, s_1, s_2) \\
\texttt{fby1}^\sharp(v, v_1.s_1, v_2.s_2) & = v.\texttt{fby1}^\sharp(v_2, s_1, s_2)
\end{array}$$

Here again, the arguments and the result of the `fby` operator must have the same clock. A `fby` is a two-state machine: while its two arguments are initially absent, it returns an absent value and remains in the initial state ($\texttt{fby}^\sharp$). When both are present, it returns the value of its first argument and enters the steady state ($\texttt{fby1}^\sharp$) which stores the previous value of its second argument, emitting it whenever both arguments are present.

$$(\texttt{fby}) : \forall cl : Clock.\ cl \times cl \to cl \text{ clock signature}$$

**Remark 2.3** (Is `fby` length preserving?)**.** It may be surprising to consider that `fby` is a length preserving function. In particular, if its second argument is empty but not the first one, it is able to return a value. But if its first argument is the empty sequence, its output is also empty.

The clock type signature does not express that the output at instant $n$ does not depend on the second input at instant $n$. Hence, both the following two equations are well clocked:

$$x = x + 1 \quad \text{or} \quad x = 0\ \texttt{fby}\ (x + 1)$$

The causality information could be embedded in the clock type system as in [26], in the case of a simple Lustre-like language (or systems with guarded types) but this calls either for adding subtyping constraints or explicit conversions. This make the clock calculus more complicated or leads to programs, in the case of a Lustre-like language, that are inelegant and not very modular.

In Lustre, Scade 6 and Lucid Synchrone, the detection of instantaneous dependences is ensured by the causality analysis, which is performed after the clock calculus. The consequence is that some valid programs cannot be written. The Signal language mixes clock inference and causality analysis [1].

We now consider the filtering (sampling) operator `when` and the combination operator `merge`.

$$\begin{array}{ll}
\texttt{when}^\sharp(abs.s_1, abs.w) & = abs.\texttt{when}^\sharp(s_1, w) \\
\texttt{when}^\sharp(v_1.s_1, \texttt{1}.w) & = v_1.\texttt{when}^\sharp(s_1, w) \\
\texttt{when}^\sharp(v_1.s_1, \texttt{0}.w) & = abs.\texttt{when}^\sharp(s_1, w) \\
\texttt{whenot}^\sharp(abs.s_1, abs.w) & = abs.\texttt{whenot}^\sharp(s_1, w) \\
\texttt{whenot}^\sharp(v_1.s_1, \texttt{0}.w) & = v_1.\texttt{whenot}^\sharp(s_1, w) \\
\texttt{whenot}^\sharp(v_1.s_1, \texttt{1}.w) & = abs.\texttt{whenot}^\sharp(s_1, w)
\end{array}$$

$$\begin{array}{ll}
\texttt{merge}^\sharp(abs.w, abs.s_1, abs.s_2) & = abs.\texttt{merge}^\sharp(w, s_1, s_2) \\
\texttt{merge}^\sharp(\texttt{1}.w, v_1.s_1, abs.s_2) & = v_1.\texttt{merge}^\sharp(w, s_1, s_2) \\
\texttt{merge}^\sharp(\texttt{0}.w, abs.s_1, v_2.s_2) & = v_2.\texttt{merge}^\sharp(w, s_1, s_2)
\end{array}$$

The result of the sampling operator `when` is present only when its first input is present and the sampling condition is present and true. The definition of `merge` says that the first branch must be present and the second must be absent when the condition is true; the first branch must be absent and the second present when the condition is false. Again, some rules are lacking. What is the clock of the result?

We need to define an operator on clocks.

$$\begin{array}{ll}
cl \,\mathtt{on}\, c & = \epsilon \text{ if } cl = \epsilon \text{ or } c = \epsilon \\
(\mathtt{1}.cl) \,\mathtt{on}\, (\mathtt{1}.c) & = \mathtt{1}.(cl \,\mathtt{on}\, c) \\
(\mathtt{1}.cl) \,\mathtt{on}\, (\mathtt{0}.c) & = \mathtt{0}.(cl \,\mathtt{on}\, c) \\
(\mathtt{0}.cl) \,\mathtt{on}\, (abs.c) & = \mathtt{0}.(cl \,\mathtt{on}\, c)
\end{array}$$

Using it, the clock type of `when` and `merge` can be expressed as:

$$\begin{array}{ll}
\mathtt{when} & : \forall cl. \forall x : cl. \forall c : cl.\ cl \,\mathtt{on}\, c \\
\mathtt{merge} & : \forall cl. \forall c : cl. \forall x : cl \,\mathtt{on}\, c. \forall y : cl \,\mathtt{on}\, (\mathtt{\textit{not}}\ \ c).\ cl
\end{array}$$

The first signature says that, for any clock $cl$, if the first input of `when` is $x$ and it has clock $cl$, the second input $c$ has clock $cl$, then the result of $x$ `when` $c$ has clock $cl \,\mathtt{on}\, c$. The rule for `whenot` is similar. The signature for `merge` says that if the first input $c$ has clock $cl$, the second input $x$ has clock $cl \,\mathtt{on}\, c$ and third input $y$ has clock $cl \,\mathtt{on}\, (\mathtt{\textit{not}}\ \ c)$, then the result of `merge` $c\ x\ y$ has clock $cl$.

The last operator we consider is the buffer. As for the definition of `const`, the production or not of a value by the operator `buffer` depends on the environment. The definition is given in Figure 11. The first parameter $(s)$ of the operator is the contents of the buffer, the second $(n)$ is the number of places remaining in the buffer, the third is the input stream, and the fourth is the clock $(w)$ of the output. The semantics only gives a meaning to programs that use bounded buffers. The operator returns a value when the output clock is `1`, provided that there is at least one stored value or an input value, and it stores input values as they arrive, provided that the number of remaining places is greater than zero. Moreover, it is not possible to store a value when the buffer is full, nor to pop a value when the buffer is empty.

The rule must be completed to deal with the empty sequence $\epsilon$. As for the Kahn semantics, the operators $op^\sharp$, $\mathtt{when}^\sharp$, $\mathtt{whenot}^\sharp$ and $\mathtt{merge}^\sharp$ return $\epsilon$ if one of their argument is $\epsilon$: $\epsilon$ is absorbing. The definitions for the operators `fby` and `buffer` applied to at least one $\epsilon$ argument are:

$$\begin{array}{llll}
\mathtt{fby}^\sharp(\epsilon, s_2) & = \epsilon & \quad \mathtt{fby1}^\sharp(\epsilon, s_1, s_2) & = \epsilon \\
\mathtt{fby}^\sharp(v_1.s_1, \epsilon) & = v_1.\epsilon & \quad \mathtt{fby1}^\sharp(v, \epsilon, s_2) & = v.\epsilon \\
\mathtt{fby}^\sharp(abs.s_1, \epsilon) & = abs.\epsilon & \quad \mathtt{fby1}^\sharp(v, s_1, \epsilon) & = v.\epsilon
\end{array}$$

$$\mathtt{buffer}^\sharp(s, n, \epsilon, w) = \epsilon$$

All these functions on clocked streams are continuous. In particular, the function $\mathtt{buffer}^\sharp$ is monotonic: given a memory $s$ and a number of remaining cells $n$

(two parameters which are not inputs of the program), for any pair of inputs $(s_1, w)$ and $(s'_1, w')$ such that $s_1 \leq s'_1$ and $w \leq w'$, we have $\texttt{buffer}^\sharp(s, n, s_1, w) \leq \texttt{buffer}^\sharp(s, n, s'_1, w')$. Continuity follows because $\texttt{buffer}^\sharp$ is length preserving.

The semantics is not directly defined on the language kernel but on a slight variation where each constant takes an extra argument specifying the clock of its result. The buffer operator also takes extra arguments: one giving the clock of its input — when a value must be pushed, — another giving the clock of its output — when a value must be popped, — and another for the size of the buffer. The following translation defines the passage from the source language:

$$\begin{array}{ll} i & \longrightarrow \texttt{const}(i, w) \\ \texttt{buffer}\ (e) & \longrightarrow \texttt{buffer}\ (n, e', w) \text{ where } e \longrightarrow e' \end{array}$$

The semantics for expressions, equations and programs are defined in the same way as for the Kahn semantics, except for constants and the buffer for which we take:

$$\begin{array}{ll} [\![\texttt{const}(i, w)]\!]_\rho^{\textsf{abs}} & = \texttt{const}^\sharp(i, w) \\ [\![\texttt{buffer}(n, e, w)]\!]_\rho^{\textsf{abs}} & = \texttt{buffer}^\sharp(\epsilon, n, [\![e]\!]_\rho^{\textsf{abs}}, w) \end{array}$$

These operators produce or not according to the operators that consume their output. This is why we add an extra argument giving the expected clock of the result. Moreover the `buffer` operator is initialized with an empty memory (written $\epsilon$). The maximum size $n$ of this memory is synthesized by the clock calculus and passed as an extra argument.

*Checking Synchrony* The example given in Figure 10 is now wrongly typed according to the composition of operator typing rules. Let *half* be the infinite periodic sequence $\texttt{1.0.1.0}\ldots = \texttt{(1.0)}$. To fulfil the typing rule for pointwise function applications, the expression $x \,\&(x \texttt{ when } \mathit{half})$ is only correct if the clock of $x$, say $cl$, equals the clock of $x \texttt{ when } \mathit{half}$, that is $cl \,\texttt{on}\, \mathit{half}$. This is impossible and this program must thus be rejected. The compiler for Lucid Synchrone [31] emits the error message:

```
let node even x = x when half
let node non_synchronous x = x & (even x)
                             ^^^^^^^^^^^^

This expression has clock 'a on half,
but is used with clock 'a
```

In the kernel language we consider, every stream $s$ is associated to a boolean sequence or *clock* with value `1` at the instants where $s$ is present and `0` otherwise. Two streams can be composed (e.g., added together) without any buffer when their clocks are equal. This is essentially a typing problem [11]. As we mentionned, it was later formulated as a *shallow embedding* in Coq, showing that clock type verification could be implemented by Coq type verification.

The successive versions of Lucid Synchrone experimented with different extensions of the initial type system. We realised that having a powerful equivalence between expressions when comparing clock types $c \,\texttt{on}\, e_1$ and $c' \,\texttt{on}\, e_2$ was not very useful. In version 2, we experimented with a very simple clock calculus reminiscent of the simple ML type system with polymorphism but extended with the rule of Laufer and Odersky [23] for existential quantification [14]. This was the basis of the clock calculus used in the Scade 6 language. Clocks are used to generate efficient imperative code, in particular to factorise control structures by grouping computations that are activated on the same clock.

It would certainly be possible to consider a shallow embedding of a language of streams together with its clock constraints using the Generalized Abstract Data Types (GADTs) of OCaml. We do not know if such an experiment has been completed.

In essence, the rule for typing an expression $e_1 + e_2$ is:

$$\frac{H \vdash e_1 : ck \qquad\qquad H \vdash e_2 : ck}{H \vdash e_1 \texttt{+} e_2 : ck}$$

This rule states that under the typing environment $H$, if $e_1$ has type $ck$ and if $e_2$ has type $ck$, then $e_1 \texttt{+} e_2$ has type $ck$. Recall that a clock type for a stream is of the form:

$$ck :: \alpha \mid ck \texttt{ on } e$$

where $\alpha$ is a clock variable and $e$ is a boolean expression. In the synchronous case, $ck_1 \texttt{ on } e_1 = ck_2 \texttt{ on } e_2$ if $ck_1 = ck_2$ and $e_1 = e_2$. Equality of types ensures equality of clocks. Hence, the composition of two flows of the same type can be defined without buffering.

## *2.2. From synchrony to n-synchrony*

In Lustre and its relatives, two input streams can be composed with a point-wise operator only when they have the same clock. This ensures that no buffer is need for the composition. This is quite constraining for video applications that are easy to describe as a Kahn process networks. If a buffer is needed, a synchronous compiler is of any help: the place where to put the buffer, its size, its input and output clock of the buffer must be determined by the programmer.

Consider for example a Picture-in-Picture as depicted in figure 12 which incrusts an image into another one. This kind of system is well modeled as a Kahn process network but the manual computation of buffer sizes is mostly manual and difficult to determine.

The PiP takes a high definition image (1920×1080 pixels), downscales it into and small definition image (720×480 pixels); it takes an other high definition image and merges it with the small definition one. The downscaler introduces a delay, hence a buffer is needed for the second image. We would like that this size be computed automatically as well as the delay (latency) for the first pixer of the output image to come up.

Can we compose non strictly synchronous streams provided their clocks are closed from each other? Can we allow for the communication between systems which are "almost" synchronous, e.g., for modeling bounded jittering or bounded

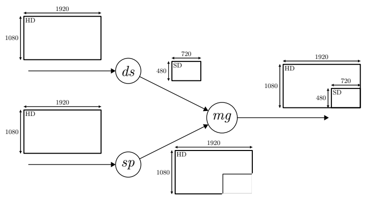


**Figure 12.** Picture in Picture

delays? Can we relax the clocking rule to give more freedom to the compiler so that it can generate more efficient code, translate into regular synchronous code if necessary?

The $n$-synchronous model [12] relaxes the classical constraints of a synchronous language like to allow for the composition of streams whose clocks are not equal but can be synchronized through the introduction of a bounded buffer. It is obtained by relaxing the clock calculus with a subtyping rule. If a stream $x$ with type $ck$ can be consumed later with type $ck'$ using a bounded buffer, we shall say that $ck$ is a subtype of $ck'$ and we write $ck <: ck'$. This allow to type a synchronous language extended with a `buffer` construct which indicates the points where the subtyping rule should be applied.

$$\frac{H \vdash e : ck \qquad ck <: ck'}{H \vdash \texttt{buffer}\ e : ck'}$$

In terms of sequences of values, `buffer` $e$ is equivalent to $e$ but it may delay its input using a bounded buffer. The `buffer` construct gives more freedom to the designer while preserving an execution in bounded memory.

Here, we consider a simple definition for $<:$ allowing to compare two types if they are of the form $\alpha$ `on` $w_1$ and $\alpha$ `on` $w_2$ only, with $w_1 <: w_2$. $w_1$ and $w_2$ are two boolean expressions.

**Definition 2.1** (Ultimately periodic clocks)**.** We consider a particular clock language $ce$ that define ultimately periodic boolean sequences only:

$$\begin{array}{lcl} ce & ::= & c \mid u\texttt{(}v\texttt{)} \\ u & ::= & \varepsilon \mid \texttt{0}.u \mid \texttt{1}.u \\ v & ::= & \texttt{0} \mid \texttt{1} \mid \texttt{0}.v \mid \texttt{1}.v \end{array}$$

It can be a variable name ($c$) or a periodic word ($u\texttt{(}v\texttt{)}$) made of a finite prefix ($u$) followed by the infinite repetition of a binary word ($v$). For example, `(10)` defines the half sequence `101010`...

### *2.2.1. Clock Adaptability*

Here is the intuition of adaptability: *a clock $w_1$ is adaptable to clock $w_2$ if any stream with clock $w_1$ can be consumed with clock $w_2$ up to the insertion of a bounded buffer.*

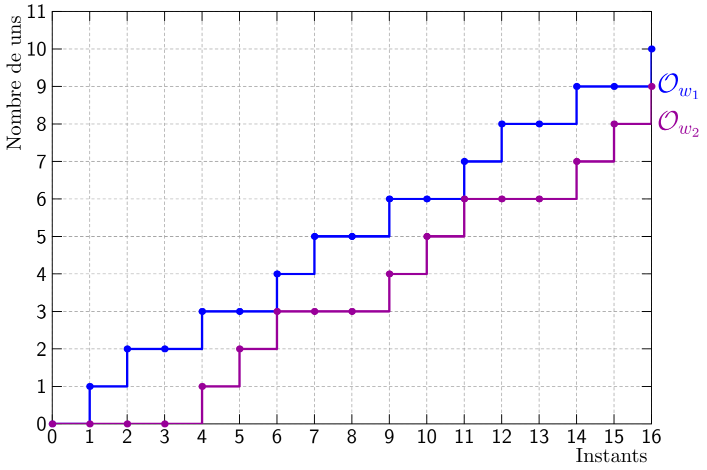


**Figure 13.** Cumulative functions for $w_1 = \texttt{(11010)}$ and $w_2 = \texttt{0(00111)}$.

To properly define this relation, we introduce the *cumulative function* of a binary word: for any binary word $w$, $\mathcal{O}_w(i)$ counts the number of 1s up to the index $i$. Figure 13 shows the cumulative functions of $w_1 = \texttt{(11010)}$ and $w_2 = \texttt{0(00111)}$.

**Definition 2.2** (Elements and Cumulative Function of $w$).
Let $w = b.w'$ with $b \in \{\texttt{0}, \texttt{1}\}$. We write $w[i]$ for the $i$-th element of $w$:

$$w[1] \overset{def}{=} b$$
$$\forall i > 1.\ w[i] \overset{def}{=} w'[i-1]$$

We write $\mathcal{O}_w$ for the cumulative function of $w$:

$$\mathcal{O}_w(0) \overset{def}{=} 0$$
$$\forall i \geq 1.\, \mathcal{O}_w(i) \overset{def}{=} \begin{cases} \mathcal{O}_w(i-1) & \text{if } w[i] = \texttt{0} \\ \mathcal{O}_w(i-1) + 1 & \text{if } w[i] = \texttt{1} \end{cases}$$

Adaptability is the conjunction of two relations: *precedence* and *synchronizability*. Precedence ensures that there is no read in an empty buffer, that is at each instant, more values have been written than read in the buffer. Synchronizability ensures that the number of values present in the buffer during the execution is bounded.

**Definition 2.3** (Synchronizability $\bowtie$, Precedence $\preceq$, Adaptability $<:$).

$$w_1 \bowtie w_2 \overset{def}{\Leftrightarrow} \exists b_1, b_2 \in \mathbb{Z}, \forall i \geq 0.\, b_1 \leq \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i) \leq b_2$$
$$w_1 \preceq w_2 \overset{def}{\Leftrightarrow} \forall i > 0.\, \mathcal{O}_{w_1}(i) \geq \mathcal{O}_{w_2}(i)$$
$$w_1 <: w_2 \overset{def}{\Leftrightarrow} w_1 \preceq w_2 \ \wedge \ w_1 \bowtie w_2$$

In Figure 13, $w_1 \bowtie w_2$ since the vertical distance between the two curves is bounded and $w_1 \preceq w_2$ since the curve $\mathcal{O}_{w_1}$ is always above the one of $\mathcal{O}_{w_2}$.

*Buffer Size.* Consider a buffer with an input clock $w_1$ and output clock $w_2$. For every instant $i$, the number of elements present in the buffer is:

$$size_i(w_1, w_2) = \mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i)$$

A negative value means that there were more reads than writes and this case should not appear. A sufficient size for the buffer is the maximal number of values present in the buffer during the execution:

$$size(w_1, w_2) = \max_{i \geq 1}(\mathcal{O}_{w_1}(i) - \mathcal{O}_{w_2}(i))$$

Thus, if $w_1$ is adaptable to $w_2$, a stream with clock $w_1$ can be safely consumed on the clock $w_2$ by insertion of a bounded buffer. Otherwise, the size of the buffer may be infinite.

The purpose of the extended clock calculus is to check that bounds exist for buffer sizes and to compute them. To this aim, subtyping constraints have to be solved and it can be done for clock that are ultimately periodic (see 2.1) [12].

To reduce the algorithmic complexity of constraint resolutions and deal with non periodic clocks, it is possible to reason with *clock envelopes*. These clock envelopes are sets of concrete clocks which are not necessarily periodic. They can model various features that exist in embedded systems such as bounded jittering, logical execution time (lower and upper bounds on the numbers of atomic steps done by a process), latencies (between when an input data is read and a output is produced), scheduling resources (a process is activated a certain number of time during a period) and the communication through buffers. Said differently, an envelope is an over abstraction of the exact clocks of the system. Hence, instead of comparing two exact clocks, we compare envelopes.

The abstraction introduced in [13] consists in reasoning on sets of clocks (or *envelopes*) defined by an asymptotic rate and two shifts bounding the potential delay with respect to this rate. It was made more precise (in the sense that it over approximates less) in [25]. Then, subtyping constraints can be replaced by linear constraints on those rates and shifts, and solved with a tool such as Glpk. We only give here an intuition of this abstraction. It was implemented for a language called Lucy-n that includes an explicit `buffer` construction and whose syntax and semantics is exactly that of the language introduced in 2. On several examples such as the *Picture in Picture*, the over-estimation due to the abstraction is small with respect to the exact solution.

### *2.2.2. Abstraction of Binary Words*

The idea behind abstraction is to reason on sets of binary words. An abstraction bounds the cumulative function of a set of words by two linear curves with the same slope. Thus, the abstraction of an infinite binary word $w$ keeps only the asymptotic proportion $r$ of `1`s in $w$ and two values $b^0$ and $b^1$ which give the minimum and maximum shift of `1`s in $w$ compared to $r$. This abstract information is called an *envelope* and noted $\langle b^0, b^1 \rangle\,(r)$.

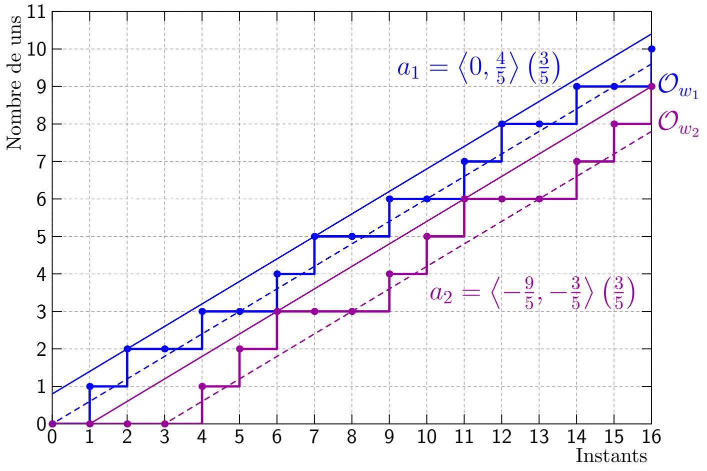


**Figure 14.** Envelopes of $w_1$ and $w_2$.

**Definition 2.4** (Concretization).

$$concr\left(\left\langle b^0, b^1 \right\rangle (r)\right) \overset{def}{=} \left\{ w \mid \forall i \geq 1, \quad \wedge \begin{array}{l} w[i] = 1 \Rightarrow \mathcal{O}_w(i) \leq r \times i + b^1 \\ w[i] = 0 \Rightarrow \mathcal{O}_w(i) \geq r \times i + b^0 \end{array} \right\}$$

with $b^0, b^1, r \in \mathbb{Q}$ and $0 \leq r \leq 1$.

The words $w_1 = \texttt{(11010)}$ and $w_2 = \texttt{0(00111)}$ seen previously are respectively in envelopes $a_1 = \left\langle 0, \frac{4}{5} \right\rangle \left(\frac{3}{5}\right)$ and $a_2 = \left\langle -\frac{9}{5}, -\frac{3}{5} \right\rangle \left(\frac{3}{5}\right)$ shown in Figure 14. In chronograms, an abstract value $\left\langle b^0, b^1 \right\rangle (r)$ is represented by two lines $\Delta^1 : \ r \times i + b^1$ and $\Delta^0 : \ r \times i + b^0$ that bound the cumulative functions of a set of binary words. The definition states that any rising edge must be below the line $\Delta^1$ (solid line) and any absence of a rising edge must be above the line $\Delta^0$ (dashed line).

For the set of words defined by an envelope to be non-empty, the line $\Delta^1$ must be above the line $\Delta^0$. At each instant, there must be a discrete value between the two lines. It is the case if the distance between them respects the following constraint.

**Proposition 2.1** (Non-empty envelope).

$$\forall a = \left\langle \frac{k^0}{\ell}, \frac{k^1}{\ell} \right\rangle \left(\frac{n}{\ell}\right) . \frac{k^1}{\ell} - \frac{k^0}{\ell} \geq 1 - \frac{1}{\ell} \Rightarrow concr\,(a) \neq \varnothing$$

The abstraction of a periodic binary word can be computed automatically.

**Definition 2.5** (Abstraction of a Periodic Word).

Let $p = u(v)$ a periodic binary word. We define $abs\,(p) \overset{def}{=} \left\langle b^0, b^1 \right\rangle (r)$ with:

$$\begin{aligned} r &= rate(p) \ = \ \tfrac{|v|_1}{|v|} \\ b^0 &= \min_{i=1..|u|+|v| \text{ with } p[i]=0} \left(\mathcal{O}_p(i) - r \times i\right) \\ b^1 &= \max_{i=1..|u|+|v| \text{ with } p[i]=1} \left(\mathcal{O}_p(i) - r \times i\right) \end{aligned}$$

where $|u|$ is the length of $u$ and $|u|_1$ its number of `1`s.

The asymptotic rate $r$ corresponds to the ratio between the number of `1`s in the periodic pattern and its length. To compute $b^0$ and $b^1$, the word must be traversed. The shift $b^0$ is the minimum difference when a `0` occurs between the number of `1`s seen at instant $i$ and the ideal value $r \times i$. The shift $b^1$ is the maximal difference between these values when a `1` occurs.

The interest of the abstraction is to reduce the complexity of exact computations and decisions on binary words by transforming them into arithmetic manipulations on rational numbers. For example, the computation of ***on*** on envelopes only needs three multiplications and two additions:

**Definition 2.6** (***on***$^\sim$ Operator)**.** Let ${b^0}_1 \leq 0$ and ${b^0}_2 \leq 0$.[10] We define:

$$\begin{array}{ll} & \langle\ {b^0}_1\ ,\ {b^1}_1\ \rangle\ (\ r_1\ ) \\ \textit{on}^\sim & \langle\ {b^0}_2\ ,\ {b^1}_2\ \rangle\ (\ r_2\ ) \\ \overset{def}{=} & \langle\ {b^0}_1 \times r_2 + {b^0}_2\ ,\ {b^1}_1 \times r_2 + {b^1}_2\ \rangle\ (\ r_1 \times r_2\ ) \end{array}$$

The elements of $w_1$ ***on*** $w_2$ are the elements of $w_1$ filtered by the elements of $w_2$. The rate of `1` in $w_1$ ***on*** $w_2$ is thus the product of the rate of $w_1$ and the one of $w_2$. When $w_1$ is sampled by $w_2$, its shifts are multiplied by $r_2$. The shifts of $w_2$ are added to those of $w_1$.

All the proofs on algebraic properties of binary sequences and abstractions have been done in Coq [25] and are available publicly. Full proofs on paper are available in the PhD. thesis of Florence Plateau [30].

## 3. Conclusion

In these course notes, we considered a simple first-order functional language that manipulates streams and functions that transform streams into streams. This language is reminiscent of the language Lustre invented by Caspi and Halbwachs, which was the basis for the development of the industrial language and environment SCADE, now regularly used in the development of critical control software, as well as the academic language Lucid Synchrone.

We showed that this language corresponds to a particular kind of Kahn process network that can be executed synchronously. This is expressed by associating a clock to every stream to indicate when the current value is present or not. Stream functions must then fullfil certain static rules to ensure that when a value is expected to be present (or absent), it is indeed present (or absent). Clocks can be understood as types and the associated static constraints as typing constraints in a type system with dependent types. Finally, we relax the synchronous constraint to allow communications through bounded buffers by adding sub-typing rules for when buffers are used. A relaxed clock calculus can infer the size of these communication buffers.

[10]We can always lose precision on the envelopes to satisfy this condition. More details are given in [30].

These course notes are far from exhaustive. In particular, they do not detail the actual clock calculus for the language, and notably the restrictions made in both Lucid Synchrone and SCADE about clock equality. In these two languages, the clock language $ce$ is limited essentially to names so that clock equality reduces to name equality. These notes also sweep under the carpet the important question of causality. E.g., equations like $x = x$ or $x = x+1$ are perfectly valid from a clock calculus point-of-view but must be rejected because $x$ depends instantaneously on itself and no sequential code can be generated: we say that $x$ is not causal. The detection of instantaneous loops or dependencies can also be handled by static typing. Finally, these notes did not address the important question of generating sequential code. Clocks are also fundamental to code generation [16]. The clock constraints can be interpreted as dedicated techniques to ensure the perfect fusion of all the intermediate streams.

*Acknowledgment*

I warmly thank Peter Muller and Alexander Pretschner for their remarkable organisation of the 2019 Marktoberdorf summer school and the atmosphere they created during the moments of work and relaxation; the invited speakers who gave splendid lectures that were all different in style; the students for their very relevant and stimulating questions. I also warmly thank Timothy Bourke for his careful reading and the suggestions he made for improving these notes.

## References

[1] T. Amagbegnon, L. Besnard, and P. Le Guernic. Implementation of the data-flow synchronous language signal. In *Programming Languages Design and Implementation (PLDI)*, pages 163–173. ACM, 1995.

[2] E. A. Ashcroft and W. W. Wadge. *Lucid, the data-flow programming language.* A.P.I.C. Studies in Data Processing, Academic Press, 1985.

[3] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), January 2003.

[4] A. Benveniste, P. Caspi, R. Lublinerman, and S. Tripakis. Actors without directors: a kahnian view of heterogeneous systems. Technical report, Verimag, Centre Équation, 38610 Gières, September 2008. Extended version of HSCC'10.

[5] A. Benveniste, P. LeGuernic, and Ch. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.

[6] G. Berry. Real time programming: Special purpose or general purpose languages. *Information Processing*, 89:11–17, 1989.

[7] G. Berry and G. Gonthier. The Esterel synchronous programming language, design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[8] Sylvain Boulmé and Grégoire Hamon. Certifying Synchrony for Free. In *International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 2250, La Havana, Cuba, December 2001. Lecture Notes in Artificial Intelligence, Springer-Verlag. Short version of *A clocked denotational semantics for Lucid-Synchrone in Coq*, available as a Technical Report (LIP6), at `www.di.ens.fr/~pouzet/bib/bib.html`.

[9] Joseph T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model.* PhD thesis, EECS Department, University of California, Berkeley, 1993.

[10] P. Caspi. Clocks in dataflow languages. *Theoretical Computer Science*, 94:125–140, 1992.

[11] Paul Caspi and Marc Pouzet. Synchronous Kahn Networks. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Philadelphia, Pensylvania, May 1996.

[12] Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. $N$-Synchronous Kahn Networks: a Relaxed Model of Synchrony for Real-Time Systems. In *ACM International Conference on Principles of Programming Languages (POPL'06)*, Charleston, South Carolina, USA, January 2006.

[13] Albert Cohen, Louis Mandel, Florence Plateau, and Marc Pouzet. Abstraction of Clocks in Synchronous Data-flow Systems. In *The Sixth ASIAN Symposium on Programming Languages and Systems (APLAS)*, Bangalore, India, December 2008.

[14] Jean-Louis Colaço and Marc Pouzet. Clocks as First Class Abstract Types. In *Third International Conference on Embedded Software (EMSOFT'03)*, Philadelphia, Pennsylvania, USA, october 2003.

[15] Jean-Louis Colaco, Bruno Pagano, and Marc Pouzet. Scade 6: A Formal Language for Embedded Critical Software Development. In *Eleventh International Symposium on Theoretical Aspect of Software Engineering (TASE)*, Sophia Antipolis, France, September 13-15 2017. Invited paper.

[16] Gwenael Delaval, Alain Girault, and Marc Pouzet. A Type System for the Automatic Distribution of Higher-order Synchronous Dataflow Programs. In *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Tucson, Arizona, June 2008.

[17] Adrien Guatto. *A Synchronous Functional Language with Integer Clocks.* PhD thesis, École normale supérieure, École normale supérieure, 45 rue d'Ulm, 75230 Paris, France, 7 janvier 2016.

[18] Adrien Guatto. A generalized modality for recursion. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 482–491, 2018.

[19] N. Halbwachs. Synchronous programming of reactive systems, a tutorial and commented bibliography. In *Tenth International Conference on Computer-Aided Verification, CAV'98*, Vancouver (B.C.), June 1998. LNCS 1427, Springer Verlag.

[20] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[21] Gilles Kahn. The semantics of a simple language for parallel programming. In *IFIP 74 Congress.* North Holland, Amsterdam, 1974.

[22] Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. In *IFIP Congress*, pages 993–998, 1977.

[23] Konstantin Läufer and Martin Odersky. An extension of ML with first-class abstract types. In *ACM SIGPLAN Workshop on ML and its Applications, San Francisco, California*, pages 78–91, June 1992.

[24] E. Lee and D. Messerschmitt. Synchronous dataflow. *IEEE Trans. Comput.*, 75(9):1235–1245, 1987.

[25] Louis Mandel, Florence Plateau, and Marc Pouzet. Lucy-n: a n-Synchronous Extension of Lustre. In *10th International Conference on Mathematics of Program Construction (MPC'10)*, Manoir St-Castin, Québec, Canada, June 2010. Springer LNCS.

[26] Louis Mandel, Florence Plateau, and Marc Pouzet. Static Scheduling of Latency Insensitive Designs with Lucy-n. In *International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, Austin, Texas, USA, October 30 – November 2 2011.

[27] T. M. Parks, J. L. Pino, and E. A. Lee. A comparison of synchronous and cycle-static dataflow. In *ASILOMAR '95: Proceedings of the 29th Asilomar Conference on Signals, Systems and Computers (2-Volume Set)*, page 204, Washington, DC, USA, 1995. IEEE Computer Society.

[28] Thomas Martyn Parks. *Bounded scheduling of process networks.* PhD thesis, EECS Department, University of California, Berkeley, Berkeley, CA, USA, 1995.

[29] Christine Paulin-Mohring. A constructive denotational semantics for Kahn networks in Coq. In Yves Bertot, Gérard Huet, Jean-Jacques Lévy, and Gordon Plotki, editors, *From Semantics to Computer Science*, pages 383–413. Cambridge University Press, 2009.

[30] Florence Plateau. *Modèle n-synchrone pour la programmation de réseaux de Kahn à mémoire bornée.* PhD thesis, Université Paris-Sud 11, Orsay, France, 6 janvier 2010.

[31] Marc Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual.* Université Paris-Sud, LRI, April 2006.

[32] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *International Conference on Compiler Construction*, 2002.

## A. Some examples

**Question A.1** (Reasonning about processes)**.** As motivated by Kahn in [21], the denotational interpretation of processes as stream function can be used to reason and prove properties about stateful systems. Consider the example in Figure 15.

- Propose an interpretation for processes `p`, `q`, `m` and `main` as continuous functions. Propose an alternative implementation of the network that use, for example, the primitives given in Figure 2. How would you prove it equivalent to the initial one?
- What does it change to remove line `(* init *)`?
- Prove that the program `main` is non blocking, i.e, if input `x` is an infinite stream, `z` is an infinite stream. It can be done with a length argument, taking $|\epsilon| = 0$ and $|v.s| = 1 + |s|$.
- Propose a sequential, equivalent implementation of `main`, made of a single elementary process with an input channel `x` and output channel `z`.

```
type 'a buff = { push: 'a -> unit; pop: unit -> 'a }

let buffer () =
  let b = Queue.create () in
  let t = Mutex.create () in
  let push v = Mutex.lock t; Queue.push v b; Mutex.unlock t in
  let pop () = Mutex.lock t; Queue.pop b; Mutex.unlock t in
  { push = push; pop = pop }

(* Process P *)
let process_p x r y () =
  y.push 0; (* init *)
  let memo = ref 0 in
  while true do
    let v = x.pop () in
    let w = r.pop () in
    memo := if v then 0 else !memo + w;
    y.push !memo
  done

(* Process Q *)
let process_q y t z () =
  while true do
    let v = y.pop () in
    t.push v; z.push v
  done

(* Process R *)
let process_m t r () =
  while true do
    let v = t.pop () in
    r.push (v + 1)
  done

(* Put them in parallel. *)
let main x z () =
  let r = buffer () in
  let y = buffer () in
  let t = buffer () in
  ignore (Thread.create (process_p x r y) ());
  ignore (Thread.create (process_q y t z) ());
  ignore (Thread.create (process_m t r) ())
```

**Figure 15.** A simple implementation of KPN with threads in OCaml

This page intentionally left blank

*Engineering Secure and Dependable Software Systems*
*A. Pretschner et al. (Eds.)*
*IOS Press, 2019*
*© 2019 The authors and IOS Press. All rights reserved.*
*doi:10.3233/978-1-61499-977-5-241*

# Efficient Checking of Actual Causality with SAT Solving

Amjad IBRAHIM [a,1], Simon REHWALD [a] and Alexander PRETSCHNER [a]
[a] *Department of Informatics, Technical University of Munich, Germany*

**Abstract.** Recent formal approaches towards causality have made the concept ready for incorporation into the technical world. However, causality reasoning is computationally hard; and no general algorithmic approach exists that efficiently infers the causes for effects. Thus, checking causality in the context of complex, multi-agent, and distributed socio-technical systems is a significant challenge. Therefore, we conceptualize an intelligent and novel algorithmic approach towards checking causality in acyclic causal models with binary variables, utilizing the optimization power in the solvers of the Boolean Satisfiability Problem (SAT). We present two SAT encodings, and an empirical evaluation of their efficiency and scalability. We show that causality is computed efficiently in less than 5 seconds for models that consist of more than 4000 variables.

**Keywords.** accountability, actual causality, sat solving, reasoning

## 1. Introduction

Causality is a fundamental construct of human perception. Although our ability to link effects to causes may seem natural, defining what precisely constitutes a cause has baffled scholars for centuries. Early work on defining causality goes back to Hume in the eighteenth century [14]. Hume's definition hinted at the idea of counter-factual relations to infer the causes of effects. Informally, we argue, counter to the fact, that $A$ is a cause of $B$ if $B$ does not occur if $A$ no longer occurs. As Lewis noted with examples, this relation is not sufficient to deal with interdependent, multi-factorial, and complex causes [21]. Thus, the search for a comprehensive general definition of causality continues. Recently, in computer science, there have been some successful and influential efforts, by Halpern and Pearl, at formalizing the idea of counter-factual reasoning and addressing the problematic examples in philosophy literature [11].

The work by Halpern and Pearl covers two notions of causality, namely actual (token) causality and type (general) causality. Type causality is a forward-looking link that forecasts the effects of causes. It is useful for predictions in different domains like medicine [17], and machine learning applications [26]. In this paper, we focus on actual causality that is a rather retrospective linking of effects to causes. We are chiefly interested in the Halpern-Pearl (HP) definition of actual causality [10]. Causality is useful

[1]Corresponding Author: Amjad Ibrahim, Boltzmannstrae 3, 85748 Garching b. Mnchen, Germany; E-mail: ibrahim@in.tum.de.

in law [24], security [18], software and system verification [3,20], databases [22], and accountability [9,16].

In essence, HP provides a formal definition of when we can call one or more events a cause of another event in a way that captures the human intuition. There have been three versions of HP: the original (2001), updated (2005), and modified (2015) versions, the latter of which we are using. The fundamental contribution of HP is that it opens the door for embedding the ability to reason about causality into socio-technical systems that are increasingly taking control of our daily lives. Among other use cases, since actual causality can be used to answer causal queries in the postmortem of unwanted behavior, it is a vital ingredient to enable accountability. Utilizing HP in technical systems makes it possible to empower them with all the other social concepts that should also be embedded into the technical world, such as responsibility [6], blame, intention, and moral responsibility [12].

Causality checking, using any version of HP, is hard. For example, under the *modified* definition, determining causality is in general $D_1^P$-complete, and $NP$-complete for singleton causes [10]. The computational complexity led to a domain-specific (e.g., database queries, counter-examples of model checking), adapted (e.g., use lineage of queries, use Kripke structure of programs), or restricted (e.g., monotone queries, singleton causes, single-equation models) utilization of HP (details in Section.2). Conversely, brute-force approaches work with small models (less than 30 variables [13]) only. Therefore, to the best of our knowledge, there exists no comprehensive, efficient, and scalable framework for modeling and benchmarking causality checking for binary models (i.e., models with binary variables only). Consequently, no existing algorithm allows applying HP on more complex examples than the simple cases in the literature.

In this paper, we argue that an efficient approach for checking causality opens the door for new use cases that leverage the concept in modern socio-technical systems. We conceptualize a novel approach towards checking causality in acyclic binary models based on the Boolean satisfiability problem (SAT). We intelligently encode the core of HP as a SAT query that allows us to reuse the optimization power built into SAT solvers. As a consequence of the rapid development of SAT solvers ($1000X+$ each decade), they offer a promising tactic for any solution in formal methods and program analysis [25]. Leveraging this power in causality establishes a robust framework for efficiently reasoning about actual causality. Moreover, since the transformation of SAT to other logic programming paradigms like answer set programming (ASP) is almost straightforward, this paper establishes the ground to tackle more causality issues (e.g., causality inference) using combinatorial solving approaches. Therefore, this paper makes the following contributions:

– An approach to check causality over binary models. It includes two SAT-encodings that reflect HP, and two variants for optimization,
– A Java library [2] that includes the implementation of our approach. It is easily extensible with new optimizations and algorithms.
– An empirical evaluation that uses different examples to show the efficiency and scalability of our approach.

[2]available at: https://github.com/amjadKhalifah/HP2SAT1.0/

## 2. Related Work

To the best of our knowledge, no previous work has tackled the technical implementation of the (*modified*) version of HP yet. Conversely, the first *two* versions were used in different applications. Although they use different versions, we still consider them related. These applications used the definition as a refinement of other technologies. For example, in [22,23,4,27], a simplified HP (the updated version) was used to refine provenance information and explain database conjunctive query results. Theses approaches, heavily depend on the correspondence between causes and domain-specific concepts like lineage, database repairs, and denial constraints. As a simplification, the authors used a single-equation causal model based on the lineage of the query in [22,23], and no-equation model in [4,27]. The approaches also eliminate HP's treatment of preemption. Similar simplification has been made for Boolean circuits in [7].

In the context of model verification, a concept that enhances a counterexample returned by a model checker with a causal explanation based on a simplified version of the updated HP was proposed in [3]. Their domain-specific simplification comes from the fact that no dependencies between variables, and hence no equations, were required. Moreover, they used the definition of singleton causes. Similarly, in [2,19], the authors implemented different flavors of causality checking (based on the updated HP) using Bounded Model Checking to debug models of safety-critical systems. They employed SAT solving indirectly in the course of model checking. The authors stated that this approach is better in large models regarding performance than their previous work.

The common ground between all these approaches and our approach is the usage of binary models. However, they were published before the modified HP version. Hence, they used the older versions. In contrast to our approach, the previous works adapted the definition for a domain specific purpose. This was reflected in restrictions on the equations of the binary model (single-equation, independent variables), the cause (singleton), or dependency on other concepts (Kripke structures, lineage formula, counter-examples). On the contrary, we propose algorithms to compute causality on binary models, without adaptations.

Similar to our aim, [13] evaluated search-based strategies for determining causality according to the original HP definition. Hopkins proposed ways to explore and prune the search space, for computing $\vec{W}, \vec{Z}$ that were required for that version, and considered properties of the causal model that makes it more efficient for computation. The results presented are of models that consist of less than 30 variables; in contrast, we show SAT-based strategies that compute causality for models of thousands of variables.

## 3. Halpern-Pearl Definition

In this section, we introduce the latest HP. All versions of HP use variables to describe the world. Structural equations define how these variables influence each other. The variables are split into *exogenous* and *endogenous* variables. The values of the former, called a *context* $\vec{u}$, are assumed to be governed by factors that are not part of the modeled world. Consequently, exogenous variables cannot be part of a cause. The values of the endogenous variables, in contrast, are determined by the mentioned equations. Formally, we describe a causal model in Definition 1. Similar to Halpern, we limit ourselves to acyclic causal models, in which we can compute a unique solution for the equations given a context $\vec{u}$.

**Definition 1** ***Causal Model*** *is a tuple $M = (U,V,R,F)$, where*

- *$U$, $V$ are sets of exogenous variables and endogenous variables respectively,*
- *$R$ associates with $Y \in U \cup V$ a set $R(Y)$ of values,*
- *$F$ associates with $X \in V$ $F_X : (\times_{U \in U} R(U)) \times (\times_{Y \in V \setminus \{X\}} R(Y)) \to R(X)$*

Here, we define the necessary notations. A *primitive event* is a formula of the form $X = x$, for $X \in V$ and $x \in R(X)$. A sequence of variables $X_1,...,X_n$ is abbreviated as $\vec{X}$. Analogously, $X_1 = x_1,...,X_n = x_n$ is abbreviated as $\vec{X} = \vec{x}$. Variable Y can be set to value y by writing $Y \leftarrow y$ (analogously $\vec{Y} \leftarrow \vec{y}$ for vectors). $\varphi$ is a Boolean combination of primitive events. $(M,\vec{u}) \models X = x$ if the variable $X$ has value $x$ in the unique solution to the equations in $M$ in context $\vec{u}$. Intervention on a model is expressed, either by setting the values of $\vec{X}$ to $\vec{x}$, written as $[X_1 \leftarrow x_1,..,X_k \leftarrow x_k]$, or by fixing the values of $\vec{X}$ in the model, written as $M_{\vec{X} \leftarrow \vec{x}}$, which effectively replaces the equations for $\vec{X}$ by a constant equation $X_i = x_i$. So, $(M,\vec{u}) \models [\vec{Y} \leftarrow \vec{y}]\varphi$ is identical to $(M_{\vec{Y} \leftarrow \vec{y}},\vec{u}) \models \varphi$ [10]. Lastly, we use $\mapsto$ to express value substitution, e.g., $[\overrightarrow{V} \mapsto \vec{v}']F_{X_j}$ refers to the evaluation of equation $F_{X_j}$ given that the values of other variables are set to $\vec{v}'$.

**Definition 2** ***Actual Cause*** *[10]*
*$\vec{X} = \vec{x}$ is an actual cause of $\varphi$ in $(M,\vec{u})$ if the following three conditions hold:*
***AC1.*** *$(M,\vec{u}) \models (\vec{X} = \vec{x})$ and $(M,\vec{u}) \models \varphi$*
***AC2.*** *There is a set $\vec{W}$ of variables in $V$ and a setting $\vec{x}'$ of the variables in $\vec{X}$ such that if $(M,\vec{u}) \models \vec{W} = \vec{w}$, then $(M,\vec{u}) \models [\vec{X} \leftarrow \vec{x}', \vec{W} \leftarrow \vec{w}]\neg\varphi$.*
***AC3.*** *$\overrightarrow{X}$ is minimal: no subset of $\vec{X}$ satisfies AC1 and AC2.*

The HP definition is presented in Definition 2. AC1 checks that the cause and the effect occurred in the real world, i.e., in $M$ given context $\vec{u}$. AC3 is a minimality check to deal with irrelevant variables. AC2 is the core; it matches the counter-factual definition of causality. It holds if there exists a *setting* $\vec{x}'$ of the variables in $\vec{X}$ different from the *original* setting $\vec{x}$ (which led to $\varphi$ holding true) and another set of variables $\vec{W}$ that we use to *fix* variables at their original value, such that $\varphi$ does not occur. Inferring $\vec{W}$ is one source of the complexity of the definition. The role of $\vec{W}$ becomes clearer when we consider the examples in [10]. Briefly, it captures the notion of *preemption* which describes the case when one possible cause rules out the other based on, e.g., temporal factors.

Halpern [10] shows that determining causality is in general $D_1^P$-complete. The family of complexity classes $D_k^P$ was introduced, in [1], to investigate the complexity of the *original* and *updated* definitions. *AC1* can be checked in polynomial time, while *AC2* is *NP*-complete, and *AC3* is co-*NP*-complete. To prove this complexity, Halpern [10] showed that AC2 could be reduced to SAT, and AC3 to UNSAT. However, the concrete encoding was not specified.

**Example** We consider a famous example from the literature: *the rock-throwing example* [21], described as follows: Suzy and Billy both throw a rock at a bottle which shatters if one of them hits it. We know that Suzy's rock hits the bottle slightly earlier than Billy's and both are accurate throwers. Halpern models this story using the following endogenous variables: $ST$, $BT$ for “Suzy/Billy throws”, with values 0 (the person does not throw) and 1 (s/he does), similarly, $SH,BH$ for “Suzy/Billy hits”, and $BS$ for “bottle shatters”. The equations are:

- $BS$ is 1 iff one of $SH$ and $BH$ is 1, i.e., $BS = SH \vee BH$
- $SH$ is 1 iff $ST$ is 1, i.e., $SH = ST$
- $BH = 1$ iff $BT = 1$ and $SH = 0$, i.e., $BH = BT \wedge \neg SH$
- $ST$, $BT$ are set by exogenous variables, i.e., $ST = ST_{exo}; BT = BT_{exo}$

Assuming a context $\vec{u}$ that sets $ST = 1$ and $BT = 1$, the original evaluation of the model is: $BS$=1 $SH$=1 $BH$=0 $ST$=1 $BT$=1. Let us assume we want to find out whether $ST$ = 1 is a cause of $BS$ = 1. Obviously, AC1 is fulfilled. As a candidate cause, we set $ST$ = 0. A first attempt with $\vec{W} = \emptyset$ shows that AC2 does *not* hold. However, if we arbitrary take $\vec{W} = \{BH\}$, i.e., we replace the equation of $BH$ with $BH$ = 0, then AC2 holds because $BS$ = 0, and AC3 automatically holds since the cause is a singleton. Thus, $ST = 1$ is a cause of $BS = 1$.

## 4. Approach

In this section, we propose our algorithmic approaches towards the HP definition. To answer a causal question efficiently, we need to find an intelligent way to search for a $\vec{W}$ such that AC2 is fulfilled as well as to check whether AC3 holds. Therefore, we propose an approach that uses SAT-solving. We show how to encode AC2 into a formula whose (un)satisfiability and thus the (un)fulfillment of AC2 is determined by a SAT-solver. Similarly, we show how to generate a formula whose satisfying assignments obtained with a solver indicate if AC3 holds.

### *4.1. Checking AC2*

For AC2, such a formula $F$ has to incorporate (1) $\neg\varphi$, (2) a context $\vec{u}$, (3) a setting, $\vec{x}'$ for candidate cause, $\vec{X}$, and (4) all possible variations of $\vec{W}$, while still (5) keeping the semantics of the underlying model $M$. In the following, we describe the concept and, then, the algorithm that generates such a formula $F$. Since we check actual causality in hindsight, we have a situation where $\vec{u}$ and $\vec{v}$ are determined, and we have a candidate cause $\vec{X} \subseteq \vec{V}$. Thus, the first two requirements are straightforward. First, the effect $\varphi$ should not hold anymore, hence, $\neg\varphi$ holds. Second, the context $\vec{u}$ should be set to its values in the original assignment (the values $\vec{u}$ of $\vec{U}$).

Since we are treating binary models only, the setting $\vec{x}'$ (from AC2) can be tailored down to negating the original value of each cause variable. This is a result of Lemma 1, which utilizes the fact that we are considering binary variables to exclude other possible settings and define precisely the *setting* $\vec{x}'$. The proof of the Lemma is given in Appendix A. Thus, to address the third requirement, according to Lemma 1, for $\neg\varphi$ to hold, all the variables of the candidate cause $\vec{X}$ are negated.

**Lemma 1** *In a binary model, if $\vec{X} = \vec{x}$ is a cause of $\varphi$, according to Definition 2, then every $\vec{x}'$ in the definition of AC2 always satisfies $\forall i.x'_i = \neg x_i$.*

To ensure that the semantics of the model are reflected in $F$ (requirement 5), we use the logical equivalence operator ($\leftrightarrow$) to express the equations. Particularly, to represent

**Algorithm 1** Check whether AC2 holds using SAT

```
Input: causal model $M$, context $\langle U_1,\dots,U_n\rangle = \langle u_1,\dots,u_n\rangle$, effect $\varphi$, candidate cause
    $\langle X_1,\dots,X_\ell\rangle = \langle x_1,\dots,x_\ell\rangle$, evaluation $\langle V_1,\dots,V_m\rangle = \langle v_1,\dots,v_m\rangle$
 1: function FULFILLSAC2($M, \vec{U}=\vec{u}, \varphi, \vec{X}=\vec{x}, \vec{V}=\vec{v}$)
 2:     if $(M,\vec{u}) \models [\vec{X} \leftarrow \neg\vec{x}]\neg\varphi$ then return $\emptyset$
 3:     else
 4:         $F := \neg\varphi \wedge \bigwedge_{i=1\dots n} f(U_i = u_i) \wedge \bigwedge_{i=1\dots m, \nexists j \bullet X_j = V_i} \big(V_i \leftrightarrow F_{V_i} \vee f(V_i = v_i)\big)$
            $\hookrightarrow \wedge \bigwedge_{i=1\dots\ell} f(X_i = \neg x_i)$
 5: with $f(Y=y) = \begin{cases} Y, & y=1 \\ \neg Y, & y=0 \end{cases}$
 6:         if $\langle U_1 = u_1 \dots U_n = u_n, V_1 = v'_1 \dots V_m = v'_m\rangle \in SAT(CNF(F))$ then
 7:             $\vec{W} := \langle W_1,\dots,W_s\rangle$ s.t. $\forall i \forall j \bullet (i \neq j \Rightarrow W_i \neq W_j) \wedge (W_i = V_j \Leftrightarrow v'_j = v_j)$
 8:             return $\vec{W}$
 9:         else return not satisfiable
10:         end if
11:     end if
12: end function
```

the endogenous variable $V_i$ and its dependency on other variables, we use this clause $V_i \leftrightarrow F_{V_i}$. This way, we create a (sub-)formula that evaluates to true if both sides are *equivalent* in their evaluation. If we do so for all other variables (that are not affected by criteria 1-3), we ensure that $F$ is only satisfiable for assignments that respect the semantics of the model.

Finally, we need to find a possibility to account for $\vec{W}$ (requirement 4) without having to iterate over the power-set of all variables. In $F$, we accomplish this by adding a disjunction with the positive or negative literal of each variable $V_i$ to the previously described equivalence-formula, depending on whether the actual evaluation of $V_i$ was 1 or 0, respectively. Then, we can interpret $((V_i \leftrightarrow F_{V_i}) \vee (\neg)V_i)$ as "$V_i$ either follows the semantics of $M$ or takes on its original value represented as a positive or negative literal". By doing so for all endogenous variables, we allow for all possible variations of $\vec{W}$. It is worth noting that we exclude those variables that are in $\vec{X}$ from obtaining their original value, as we are already changed their setting to $\neg\vec{x}$ and thus keeping a potential cause at its original value is not reasonable. Obviously, it might not make sense to always add the original value for all variables. We leave this as a candidate optimization for a future work.

***AC2 Algorithm*** We formalize the above in Algorithm 1. The evaluation, in the input, is a list of all the variables in $M$ and their values under $\vec{u}$. The rest is self-explanatory. We slightly change the definition of $\varphi$ from a combination of primitive events to a combination of literals. For instance, instead of writing $\varphi = (X_1 = 1 \wedge X_2 = 0 \vee X_3 = 1)$, we would use $\varphi = (X_1 \wedge \neg X_2 \vee X_3)$. In other words, we replace each primitive $(X = x) \in \varphi$ with $X$ if $x = 1$ or $\neg X$ if $x = 0$ in the original assignment, such that we use $\varphi$ in a formula. The same logic is achieved using the function $f(Y = y)$ in line 5 of the algorithm.

Before we construct formula $F$, we check if $\vec{X} = \vec{x}$ given $\vec{W} = \emptyset$ (line 2) fulfills AC2. Hence, in this case, we do not need to look for a $\vec{W}$. Otherwise, we construct $F$ (line 4) that is a conjunction of $\neg\varphi$ and the exogenous variables of $M$ as literals depending on $\vec{u}$. Note that $\varphi$ does not necessarily consist of a single variable only; it can be any Boolean formula. For example, if $\varphi = (BS = 1 \land BH = 0)$ in the notation as defined by [10], we would represent it in $F$ as $(BS \land \neg BH)$. This consideration is handled by Algorithm 1 without further modification. In addition, we represent each endogenous variable, $V_i \notin \vec{X}$ with a disjunction between its equivalence formula $V_i \leftrightarrow F_{V_i}$ and its literal representation. To conclude the formula construction, we add the negation of the candidate cause $\vec{X} = \vec{x}$, a consequence of Lemma 1.

If $F$, represented in a conjunctive normal form, is satisfiable, we obtain the satisfying assignment (line 6) and compute $\vec{W}$ (line 7) as the set of those variables whose valuations were *not* changed in order to ensure $\neg\varphi$ that is finally returned. If $F$ is unsatisfiable, *not satisfiable* is returned.

***Minimality of*** $\vec{W}$ In Algorithm 1, we considered $\vec{W}$ to consist of all the variables whose original evaluation and satisfying assignments are equal. This is an over-approximation of the $\vec{W}$ set because, possibly, there are variables that are not affected by changing the values of the cause, and are yet not required to be fixed in $\vec{W}$. Despite this consideration, a non-minimal $\vec{W}$ is still valid according to HP. However, to compute the degree of responsibility [6], a minimal $\vec{W}$ is required. Therefore, we briefly discuss a modification that yields a minimal $\vec{W}$.

We need to modify two parts of Algorithm 1. First, we cannot just consider *one* satisfying assignment for $F$. Rather, we need to analyze *all* the assignments. Determining all the assignments is called an All-SAT problem. Second, we have to further analyze each assignment of $\vec{W}$ to check if we can find a subset such that $F$, and thus AC2, still holds. Specifically, we check if each element in $\vec{W}$ is equal to its original value because it was explicitly set so, or because it simply evaluated according to its equation. In the latter case, it is *not* a required part of $\vec{W}$. Precisely, in Algorithm 1, everything stays the same until the computation of $F$. After that, we check whether $F$ is satisfiable, but now we compute all the satisfying assignments. Subsequently, for each satisfying assignment, we compute $\vec{W}_i$ as explained. Then, we return the smallest $\vec{W}_i$, at the cost of iterating over *all* satisfying assignments of the variables in $V$.

## 4.2. Checking AC3

Our approach for checking AC3 using SAT is very similar to the one for AC2. We construct another SAT formula, $G$. The difference between $G$ and $F$ is in how the parts of the cause are represented. In $G$, we allow each of them to take on its original value *or* its negation (e.g., $A \lor \neg A$). Clearly, we could replace that disjunction with *true* or 1. However, we explicitly do not perform this simplification such that a satisfying assignment for $G$ still contains all variables of the causal model, $M$.

In general, the idea is as follows. If we find a satisfying assignment for $G$ such that at least one conjunct of the cause $\vec{X} = \vec{x}$ takes on a value that equals the one computed from its corresponding equation, then, we know that this particular conjunct is not required to be part of the cause and there exists a subset of $\vec{X}$ that fulfills AC2 as well. The same applies if the conjunct is equal to its original value in the satisfying assignment; this would mean that it is part of a $\vec{W}$ such that $\neg\varphi$ holds. When collecting all those conjuncts, we can construct a new cause $\vec{X}' = \vec{x}'$ by subtracting them from the original

**Algorithm 2** Check whether AC3 holds using ALL-SAT

```
Input: causal model $M$, context $\langle U_1,\dots,U_n\rangle = \langle u_1,\dots,u_n\rangle$, effect $\varphi$, candidate cause
    $\langle X_1,\dots,X_\ell\rangle = \langle x_1,\dots,x_\ell\rangle$, evaluation $\langle V_1,\dots,V_m\rangle = \langle v_1,\dots,v_m\rangle$
1: function FULFILLSAC3($M, \vec{U} = \vec{u}, \varphi, \vec{X} = \vec{x}, \vec{V} = \vec{v}$)
2:     if $\ell > 1 \wedge (M,\vec{u}) \models \varphi$ then
3:         $G := \neg\varphi \wedge \bigwedge_{i=1\dots n} f(U_i = u_i) \wedge \bigwedge_{i=1\dots m, \nexists j \bullet X_j = V_i} \big(V_i \leftrightarrow F_{V_i} \vee f(V_i = v_i)\big)$
             $\hookrightarrow \wedge \bigwedge_{i=1\dots\ell} X_i \vee \neg X_i$
4:         for all $\langle \vec{U} = \vec{u}, \vec{V} = \vec{v}'\rangle \in SAT(CNF(G))$ do
5:             if $|\Big\{ j \in \{1,..,\ell\} | \exists i \bullet V_i = X_j \wedge v'_i \neq v_i$
             $\hookrightarrow \wedge v'_i \neq [\overrightarrow{V} \mapsto \vec{v}']F_{X_j} \Big\}| < \ell$ then return false
6:             end if
7:         end for
8:     end if
9:     return true
10: end function
```

cause and then checking whether or not it fulfills AC1. If it does, AC3 is violated because we identified a subset $\vec{X}'$ of $\vec{X}$ for which both AC1 and AC2 hold.

***AC3 Algorithm*** We formalize our approach in Algorithm 2. The input and the function $f(V_i = v_i)$ remain the same as for Algorithm 1; the latter is omitted. In case $\vec{X} = \vec{x}$ is a singleton cause or $\varphi$ did not occur, AC3 is then fulfilled automatically (line 2). Otherwise, line 3 shows how formula $G$ is constructed. This construction is only different from the construction of $F$ in Algorithm 1 in how to treat variables $\in \vec{X}$. They are added as a disjunction of their positive and negative literals. Once $G$ is constructed, we check its satisfiability, if it is not satisfiable we return *true*, i.e., AC3 is fulfilled. For example, this can be the case if the candidate cause $\vec{X}$ did not satisfy AC2. Otherwise, we check *all* its satisfying assignments. We need to do this, as $G$ might also be satisfiable for the original $\vec{X} = \vec{x}$ so that we cannot say for sure that any satisfying assignment found, proves that there exists a subset of the cause. Instead, we need to obtain all of them. Obviously, this is problematic and could decrease the performance if $G$ is satisfiable for a large number of assignments. Therefore, we plan to address this in future work.

However, for now, we compute one assignment and check the *count* of the conjuncts in the cause that have different values in $\vec{v}'$ than their original, and that their formula does not evaluate to this assignment (line 5). If the *count* is less than the size of the cause, then AC3 is violated. Otherwise we check another assignment.

***Combining AC2 and AC3*** While developing Algorithm 1 and Algorithm 2, we discovered that combining both is an option for optimizing our approach. In particular, we can exploit the relationship between the satisfying assignment(s) for the formulas $F$ and $G$, i.e., $\vec{a}_F \in A_G$. This holds, as we allow the variables $\vec{X}$ of a cause to be both 1 or 0 in $G$ so that we can show that the satisfying assignment, $\vec{a}_F$ for $F$ in Algorithm 1 is an element of the satisfying assignments $A_G$, for $G$. Then, instead of computing both $F$ and $G$, we could just compute $G$, then filter those satisfying assignments that $F$ would have yielded and use them for checking AC2 while we use all satisfying assignments of $G$ to check AC3.

**Table 1.** $F$, $G$ assignments

|  | $BS$ | $SH$ | $BH$ | $ST$ | $BT$ |
|---|---|---|---|---|---|
| $M$ | 1 | 1 | 0 | 1 | 1 |
| $F$ | 0 | 0 | 0 | 0 | 1 |
| $G\,\vec{a}_1$ | 0 | 0 | 0 | 0 | 0 |
| $G\,\vec{a}_2$ | 0 | 0 | 0 | 0 | 1 |

## 4.3. Example

Recall the example from Section.3. Since the context $\vec{u}$ sets $ST = 1$ and $BT = 1$, the original evaluation of the model is shown in the first row of Table.1. We want to find out whether $ST = 1$ is a cause of $BS = 1$. Algorithm.1 generates the following $F$, that is satisfiable for one assignment (Table.1 second row): $BS = 0$, $SH = 0$, $BH = 0$, $ST = 0$, $BT = 1$. All the variables, except $BH$ and $BT$, change their evaluation. Thus, we conclude that $ST = 1$ fulfills AC2 with $\vec{W} = \{BH, BT\}$. Notice that even though this $\vec{W}$ is not minimal, it is still valid. That said, we still can calculate a minimal $\vec{W}$.

$$F = \overbrace{\neg BS}^{\neg\varphi} \wedge \overbrace{ST_{exo} \wedge BT_{exo}}^{\vec{u}} \wedge \big(\overbrace{(BS \leftrightarrow SH \vee BH)}^{\text{equation of BS}} \vee \overbrace{BS}^{\text{orig. BS}}\big) \wedge \big(\underbrace{(SH \leftrightarrow ST)}_{\text{equation of SH}} \vee \underbrace{SH}_{\text{orig. SH}}\big)$$
$$\wedge \big(\underbrace{(BH \leftrightarrow BT \wedge \neg SH)}_{\text{equation of BH}} \vee \underbrace{\neg BH}_{\text{orig. BH}}\big) \wedge \underbrace{\neg ST}_{\text{equation of ST}} \wedge \big(\underbrace{(BT \leftrightarrow BT_{exo})}_{\text{equation of BT}} \vee \underbrace{BT}_{\text{orig. BT}}\big)$$

To illustrate checking AC3, we ask a different question: are $ST = 1$ *and* $BT = 1$ a cause of $BS = 1$? Note that AC2 is fulfilled with $W = \emptyset$, for this cause. Obviously, if both do not throw, the bottle does not shatter. Using Algorithm 2, we obtain the following $G$ formula.

$$G = \overbrace{\neg BS}^{\neg\varphi} \wedge \overbrace{ST_{exo} \wedge BT_{exo}}^{\vec{u}} \wedge \big(\overbrace{(BS \leftrightarrow SH \vee BH)}^{\text{equation of } BS} \vee \overbrace{BS}^{\text{orig. } BS}\big) \wedge \big(\overbrace{(SH \leftrightarrow ST)}^{\text{equation of } SH} \vee \overbrace{SH}^{\text{orig. } SH}\big)$$
$$\wedge \big(\underbrace{(BH \leftrightarrow BT \wedge \neg SH)}_{\text{equation of } BH} \vee \underbrace{\neg BH}_{\text{orig. } BH}\big) \wedge \big(\underbrace{ST}_{\text{orig. } ST} \vee \underbrace{\neg ST}_{\text{negated orig. } ST}\big) \wedge \big(\underbrace{BT}_{\text{orig. } BT} \vee \underbrace{\neg BT}_{\text{negated orig. } BT}\big)$$

As Table.1 shows, $G$ is satisfiable with *two* assignments $\vec{a}_1$ and $\vec{a}_2$. For $\vec{a}_1$, we can see that both $ST$ and $BT$ have values different from their original evaluation, and that both do not evaluate according to their equations. Thus, we cannot show that AC3 is violated, yet. For $\vec{a}_2$, $BT = 1$, so it is equal to the evaluation of its equation. Consequently, $BT$ is not a required part of $\vec{X}$, because $\neg\varphi = \neg BS$ still holds although we did not set $BT = 0$. So, AC3 is not fulfilled because AC1 and AC2 hold for a subset of the cause.

## 5. Evaluation

In this section, we provide details on the implementation of our algorithms, and evaluate their efficiency.

**Table 2.** Evaluated causal models

| Causal Model | Endogenous Vars. |
|---|---|
| Abstract Models: $AM_1$ , $AM_2$ | 8 , 3 |
| Steal Master Key: 3 suspects ($SMK_3$), 8 suspects ($SMK_8$) | 36, 91 |
| Leakage in Sub-sea Production System (LSP) [5] | 41 |
| Binary Trees of different heights (BT) | 15 - 4095 |
| Abstract Model 1 Combined with Binary Tree (ABT) | 4103 |

## 5.1. Technical implementation

Our implementation is a Java library. As such, it can easily be integrated into other systems. It supports both the creation of binary causal models as well as solving causality problems. For the modeling part and the implementation of our SAT-based approach, we take advantage of the library *LogicNG* [3] . It provides methods for creating and modifying boolean formulas and allows to analyze those using different SAT solvers. We use the implementation of *MiniSAT* solver [8] within *LogicNG*. For the sake of this evaluation, we will compare the *execution time* and *memory allocation* for the following *four* strategies: `BRUTE_FORCE` -a standard brute-force implementation of HP, `SAT` -SAT-based approach (Algorithm1, 2), `SAT_MINIMAL` -the minimal $\vec{W}$ extension, and `SAT_COMBINED` -optimization of the SAT-based approach by combining AC2 and AC3. All our measurements were performed on Ubuntu 16.04 LTS machine equipped with an Intel® Core$^{TM}$ i7-3740QM CPU and 8 GB RAM. For each benchmark, we specified 100 warmup and 100 measurement iterations.

## 5.2. Methodology and Evaluated Models

In summary, we experimented with 12 different models. On the one hand, we took the binary models from [10]. There were 5 of them in total, namely, *Rock-Throwing*, *Forest Fire*, *Prisoners*, *Assassin*, and *Railroad*. Since these examples are rather small ($\leq 5$ variables) and easy to understand, they mainly serve for sanity checks of our approach. On the other hand, we used examples that do not stem from the literature on causality. One is a security example obtained from an industrial partner. It describes the causal factors that lead to stealing a security master key by an insider. We refer to it as *SMK*. We used two variants of that example, one with 3 suspects, and the other with 8 suspects. We also used one example from the safety domain that describes a leakage in a sub-sea production system; we refer to it as *LSP*. Last, we artificially generated models of binary trees with different heights, denoted as $BT_{depth}$, and combined them with other random models (non-tree graphs), denoted as *ABT*. For a thorough description of each model, please refer to this report [15]. Table 2 shows the list of the bigger models, along with the number of endogenous variables. In total, we analyzed 278 scenarios, however, for space limits, we focus on a subset of the scenarios.

## 5.3. Discussion and Results

In this section, we discuss some representative cases from our experiments. Table 3 shows the details of these cases. The first two columns show the scenario identifier,

[3] https://github.com/logic-ng/LogicNG

Table 3. Discussed scenarios as part of the analysis

| | ID | $\vec{X} = \vec{x}$ | $\varphi$ | AC1 | AC2 | AC3 | $\lvert\text{Minimal } \vec{W}\rvert$ | BRUTE | SAT | $\text{SAT}^{\text{MINIMAL}}$ | $\text{SAT}_{\text{COMBINED}}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $SMK_3$ | 3 | $FS_{U_1} = 1 \wedge FN_{U_1} = 1 \wedge A_{U_1} = 1$ | $SMK = 1$ | ✓ | ✓ | ✓ | 4 | N/A N/A | 0.8952ms 1.0540MB | 1.0924ms 1.2223MB | 0.7160ms 0.9032MB |
| | 22 | $FS_{U_3} = 1$ | | ✓ | ✗ | ✓ | – | N/A N/A | 0.6159ms 0.8650MB | 0.6232ms 0.8651MB | 0.6098ms 0.8614MB |
| | 24 | $FS_{U_3} = 1 \wedge FN_{U_3} = 1 \wedge A_{U_3} = 1$ | | ✓ | ✓ | ✓ | 0 | N/A N/A | 0.7132ms 0.9164MB | 0.7217ms 0.9165MB | 0.7157ms 0.9164MB |
| | 26 | $FS_{U_3} = 1 \wedge FN_{U_3} = 1 \wedge A_{U_3} = 1 \wedge AD_{U_3} = 1$ | | ✓ | ✓ | ✗ | 0 | N/A N/A | 0.7725ms 0.9577MB | 0.7887ms 0.9577MB | 0.7754ms 0.9577MB |
| | 29 | $A_{U_3} = 1 \wedge AD_{U_3} = 1$ | $SD = 1$ | ✓ | ✓ | ✗ | 0 | 0.1360ms 0.2268MB | 0.7231ms 0.9198MB | 0.7233ms 0.9233MB | 0.7225ms 0.9198MB |
| LSP | 3 | $X_1 = 1 \wedge X_2 = 1$ | $X_{41} = 1$ | ✓ | ✓ | ✗ | 0 | 0.1600ms 0.2524MB | 0.8600ms 1.0953MB | 0.8648ms 1.0903MB | 0.8598ms 1.0913MB |
| | 56 | $X_1 = 1 \wedge X_2 = 1$ | | ✓ | ✗ | ✓ | – | N/A N/A | 0.9464ms 1.2348MB | 1.0245ms 1.2308MB | 0.6984ms 0.9918MB |
| | 57 | $X_1 = 1 \wedge X_3 = 1$ | | ✓ | ✓ | ✓ | 0 | N/A N/A | 0.8032ms 1.0600MB | 0.8025ms 1.0599MB | 0.8092ms 1.0560MB |
| BT height = 12 | 34 | $n_{4093} = 1 \wedge n_{4094} = 1$ | $n_{\text{root}} = 1$ | ✓ | ✗ | ✓ | – | N/A N/A | 6540.3ms 2080MB | 6410ms 2077MB | 3587.9ms 1055.0MB |
| | 35 | $n_{4091} = 1 \wedge n_{4092} = 1 \wedge n_{4093} = 1 \wedge n_{4094} = 1$ | | ✓ | ✗ | ✓ | – | N/A N/A | 6949ms 2090.2MB | 6618ms 2090.8MB | 3328.4ms 1054.9MB |
| ABT | 1 | $n_{4094} = 1$ | $I = 1$ | ✓ | ✓ | ✓ | 4 | N/A N/A | 3948.1ms 1055.0MB | 15324ms 4019.1MB | 3824.8ms 1055.5MB |
| | 4 | $n_{4093} = 1 \wedge n_{4094} = 1$ | | ✓ | ✓ | ✓ | 4 | N/A N/A | 6379ms 2042.3MB | 19043ms 5088.6MB | 3718.5ms 1057.3MB |
| | 5 | $n_{4092} = 0 \wedge n_{4093} = 1 \wedge n_{4094} = 1$ | | ✓ | ✓ | ✗ | 5 | N/A N/A | 7906ms 2047.0MB | 19271ms 5123.2MB | 3803.3ms 1058.8MB |

namely, the *name* of the model and the *ID* of the scenario that differs in the details of the causal query, i.e., $\vec{X}$ and $\varphi$. The latter are shown in the third and fourth columns. Then, the results of the three conditions are displayed in columns AC1-AC3. The size of the minimal W set is displayed in the next column. Finally, for each algorithm, the execution time and memory allocation are shown. We write N/A in cases where the computation was not completed in 5 minutes or consumed too much memory. As a general remark, it does not matter which algorithm is applied if AC2 holds for an empty $\vec{W}$ and AC3 holds automatically, i.e., $\vec{X}$ is a singleton.

As expected, the Brute-Force approach (BF) works only for smaller models (<5 variables), or in situations where only a few iterations are required. Such as scenarios *LSP-3, and SMK-29.* Specifically, we see these situations when AC2 holds with a small or empty $\vec{W}$, and AC3 does not hold. That is, the number of iterations BF performs is small because the sets, $\vec{W}_i$ are ordered by size. Such examples did not exhibit the major problem of BF, i.e., the generation of all possible sets, $\vec{W}_i$ whose number increases exponentially, and the iterations BF might, therefore, perform to check minimality in AC3.

For larger models ( >30 variables), BF did not return an answer in 5 minutes, especially when AC2 does not hold. This is seen by the several N/A entries in Table 3. For example, in the SMK, the set of all possible $\vec{W}_i$ has a size of up to $2^{35}$. In the worst case, this number of iterations is required for finding out that AC2 does not hold. It is possible that this number of iterations multiplied by the number of subsets of the cause needs to be executed again to check AC3. This causes BF to be extremely slow and to consume a lot of memory. The SAT by contrast, always stays below 1.5 ms and allocates less than 1.5 MB during the execution for all scenarios of the SMK. Even if larger models were considered, SAT handles them efficiently, e.g., *BT 34 - 35*, where the underlying causal model contains 4095 variables, executed in $\leq 7s$. However, the latter scenarios are special because AC2 does not hold. In *ABT 1, 4 and 5*, we can see that even if AC2 *does* hold and $\vec{W}$ is *not* empty, SAT takes only $8s$.

While obtaining a minimal $\vec{W}$ using our approach showed a rather small impact relative to the SAT approach in most of the scenarios, it showed a significant increase in some scenarios. This impact was highly dependent on the nature and semantics of the underlying causal model. That is, we can only observe a major impact if the number of satisfying assignments or the size of a non-minimal $\vec{W}$ is large as this will significantly extend the analysis. For instance, in *SMK-3*, the execution time increased by about 22%. Nonetheless, there are scenarios in which we observed a significant increase, such as the *ABT-4* scenario. Here, the non-minimal $\vec{W}$ contains more than 4000 elements, leading to an increase of more than 200% in the execution time required to determine a minimal $\vec{W}$.

Finally, combining the algorithms for AC2 and AC3 is only beneficial if AC2 and AC3 need to be explicitly analyzed (AC2 does not hold for an empty W, and the cause is not a singleton). We have many such scenarios in our examples. In evaluating them, we found out that there is a positive impact in using this optimization, but it is rather small on the average. Larger differences can be seen, for instance, in *ABT-5* where the SAT-based approach executes for 7906*ms* while the current optimization takes 3803*ms*.

The main finding of our experiments is that actual causality can be computed efficiently with our SAT-based approach. Within binary models of *4000* variables, we were able to obtain a correct answer for any query in less than 4 seconds, using a memory of *1 GB*.

## 6. Conclusions and Future Work

It is difficult to devise automated assistance for causality reasoning in modern socio-technical systems. Causality checking, according to the formal definitions, is computationally hard. Therefore, efficient approaches that scale to the complexity of such systems are required. In the course of this, we proposed an intelligent way to utilize SAT solvers to check actual causality in binary models in a large scale that we believe to be particularly relevant for accountability purposes. We empirically showed that it can efficiently compute actual causality in large binary models. Even with only 30 variables, determining causality in a brute force manner is incomputable, whereas our SAT-based approach returned a result for such cases in 1 ms. In addition, causal models consisting of more than 4000 endogenous variables were still handled within seconds using the proposed approach.

For future work, we will consider other logic programming paradigms such as integer linear programming and answer set programming to develop our approach from checking to possibly inferring causality. Moreover, a thorough characterization of causal model classes that affect the efficiency of the proposed approach is a useful follow-up to this work. We plan to extend our benchmark with models of different patterns, and different cardinalities of causes.

## References

[1] Aleksandrowicz, G., Chockler, H., Halpern, J.Y., Ivrii, A.: The computational complexity of structure-based causality. In: Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada. pp. 974–980 (2014), http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8328

[2] Beer, A., Heidinger, S., Kühne, U., Leitner-Fischer, F., Leue, S.: Symbolic causality checking using bounded model checking. In: Model Checking Software - 22nd International Symposium, SPIN 2015, Stellenbosch, South Africa, August 24-26, 2015, Proceedings. pp. 203–221 (2015)

[3] Beer, I., Ben-David, S., Chockler, H., Orni, A., Trefler, R.J.: Explaining counterexamples using causality. Formal Methods in System Design **40**(1), 20–40 (2012). https://doi.org/10.1007/s10703-011-0132-2, https://doi.org/10.1007/s10703-011-0132-2

[4] Bertossi, L.: Characterizing and computing causes for query answers in databases from database repairs and repair programs. In: International Symposium on Foundations of Information and Knowledge Systems. pp. 55–76. Springer (2018)

[5] Cheliyan, A.S., Bhattacharyya, S.K.: Fuzzy fault tree analysis of oil and gas leakage in subsea production systems. Journal of Ocean Engineering and Science **3**(1), 38 – 48 (2018). https://doi.org/https://doi.org/10.1016/j.joes.2017.11.005, http://www.sciencedirect.com/science/article/pii/S2468013317300591

[6] Chockler, H., Halpern, J.Y.: Responsibility and blame: A structural-model approach. J. Artif. Intell. Res. **22**, 93–115 (2004). https://doi.org/10.1613/jair.1391, https://doi.org/10.1613/jair.1391

[7] Chockler, H., Halpern, J.Y., Kupferman, O.: What causes a system to satisfy a specification? ACM Transactions on Computational Logic (TOCL) **9**(3), 20 (2008)

[8] Eén, N., Sörensson, N.: An extensible sat-solver. In: Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers. pp. 502–518 (2003)

[9] Feigenbaum, J., Jaggard, A.D., Wright, R.N.: Towards a formal model of accountability. In: 2011 New Security Paradigms Workshop, NSPW '11, Marin County, CA, USA, September 12-15, 2011. pp. 45–56 (2011). https://doi.org/10.1145/2073276.2073282, http://doi.acm.org/10.1145/2073276.2073282

[10] Halpern, J.Y.: A modification of the halpern-pearl definition of causality. In: Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015. pp. 3022–3033 (2015), http://ijcai.org/Abstract/15/427

[11] Halpern, J.Y.: Actual causality. The MIT Press, Cambridge, Massachussetts (2016)

[12] Halpern, J.Y., Kleiman-Weiner, M.: Towards formal definitions of blameworthiness, intention, and moral responsibility. In: Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI-18) (2018)

[13] Hopkins, M.: Strategies for determining causes of events. In: AAAI/IAAI. pp. 546–552 (2002)

[14] Hume, D.: An Enquiry Concerning Human Understanding (1748)

[15] Ibrahim, A.: Efficient checking of actual causality via sat solving - benchmarked models, https://github.com/amjadKhalifah/HP2SAT1.0/blob/master/doc/models.pdf

[16] Kacianka, S., Kelbert, F., Pretschner, A.: Towards a unified model of accountability infrastructures. In: Proceedings First Workshop on Causal Reasoning for Embedded and safety-critical Systems Technologies, CREST@ETAPS 2016, Eindhoven, The Netherlands, 8th April 2016. pp. 40–54 (2016). https://doi.org/10.4204/EPTCS.224.5, https://doi.org/10.4204/EPTCS.224.5

[17] Kleinberg, S., Hripcsak, G.: A review of causal inference for biomedical informatics. Journal of Biomedical Informatics **44**(6), 1102–1112 (2011). https://doi.org/10.1016/j.jbi.2011.07.001, https://doi.org/10.1016/j.jbi.2011.07.001

[18] Künnemann, R., Esiyok, I., Backes, M.: Automated verification of accountability in security protocols. CoRR **abs/1805.10891** (2018), http://arxiv.org/abs/1805.10891

[19] Leitner-Fischer, F.: Causality Checking of Safety-Critical Software and Systems. Ph.D. thesis, University of Konstanz, Germany (2015), http://kops.uni-konstanz.de/handle/123456789/30778

[20] Leitner-Fischer, F., Leue, S.: Causality checking for complex system models. In: Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings (2013)

[21] Lewis, D.: Causation. Journal of Philosophy **70**(17), 556–567 (1973). https://doi.org/10.2307/2025310

[22] Meliou, A., Gatterbauer, W., Halpern, J.Y., Koch, C., Moore, K.F., Suciu, D.: Causality in databases. IEEE Data Eng. Bull. **33**(3), 59–67 (2010), http://sites.computer.org/debull/A10sept/suciu.pdf

[23] Meliou, A., Gatterbauer, W., Moore, K.F., Suciu, D.: The complexity of causality and responsibility for query answers and non-answers. PVLDB **4**(1), 34–45 (2010), http://www.vldb.org/pvldb/vol4/p34-meliou.pdf

[24] Moore, M.S.: Causation and responsibility : an essay in law, morals, and metaphysics. Oxford Univ. Press, Oxford (2009), http://bvbr.bib-bvb.de:8991/F?func=service&doc_library=BVB01&doc_number=016740811&line_number=0002&func_code=DB_RECORDS&service_type=MEDIA;http://bvbr.bib-bvb.de:8991/F?func=service&doc_library=BVB01&doc_number=016740811&line_number=0001&func_code=DB_RECORDS&service_type=MEDIA

[25] Newsham, Z., Ganesh, V., Fischmeister, S., Audemard, G., Simon, L.: Impact of community structure on sat solver performance. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 252–268. Springer (2014)

[26] Pearl, J.: Theoretical impediments to machine learning with seven sparks from the causal revolution. arXiv preprint arXiv:1801.04016 (2018)

[27] Salimi, B., Bertossi, L.: From causes for database queries to repairs and model-based diagnosis and back. arXiv preprint arXiv:1412.4311 (2014)

## A. Appendix: Lemma Proof

**Lemma 2** *In a binary model, if $\vec{X} = \vec{x}$ is a cause of $\varphi$, according to HP [10] definition, then every $\vec{x}'$ in the definition of AC2 always satisfies $\forall i \bullet x'_i = \neg x_i$.*

**Proof 1** *We use the following notation: $\vec{X}_{(n)}$ stands for a vector of length n, $X_1, \ldots, X_n$; and $\vec{X}_{(n)} = \vec{x}_{(n)}$ stands for $X_1 = x_1, \ldots, X_n = x_n$. Let $\vec{X}_{(n)} = \vec{x}_{(n)}$ be a cause for $\varphi$ in some model M.*

1. *AC1 yields*

$$(M, \vec{u}) \models (\vec{X}_{(n)} = \vec{x}_{(n)}) \wedge (M, \vec{u}) \models \varphi. \tag{1}$$

2. *Assume that the lemma does not hold. Then there is some index k such that $x'_k = x_k$ and AC2 holds. Because we are free to choose the ordering of the variables, let us set $k = n$ wlog. We may then rewrite AC2 as follows:*

$$\exists \vec{W}, \vec{w}, \vec{x}'_{(n)} \bullet (M, \vec{u}) \models (\vec{W} = \vec{w}) \implies (M, \vec{u}) \models \left[\vec{X}_{(n-1)} \leftarrow \vec{x}'_{(n-1)}, X_n \leftarrow x_n, \vec{W} \leftarrow \vec{w}\right] \neg\varphi. \tag{2}$$

3. *We will show that equations 1 and 2 give rise to a smaller cause, namely $\vec{X}_{(n-1)} = \vec{x}_{(n-1)}$, contradicting the minimality requirement AC3. We need to show that the smaller cause $\vec{X}_{(n-1)} = \vec{x}_{(n-1)}$ satisfy AC1 and AC2, as stated by equations 3 and 4 below. This violates the minimality requirement of AC3 for $\vec{X}_{(n)} = \vec{x}_{(n)}$.*

$$(M, \vec{u}) \models (\vec{X}_{(n-1)} = \vec{x}_{(n-1)}) \wedge (M, \vec{u}) \models \varphi \tag{3}$$

*states AC1 for a candidate "smaller" cause $\vec{X}_{(n-1)}$. Similarly,*

$$\exists \overrightarrow{W}^*, \overrightarrow{w}^*, \overrightarrow{x}'^*_{(n-1)} \bullet (M, \overrightarrow{u}) \models (\overrightarrow{W}^* = \overrightarrow{w}^*) \implies (M, \overrightarrow{u}) \models \left[\overrightarrow{X}_{(n-1)} \leftarrow \overrightarrow{x}'^*_{(n-1)}, \overrightarrow{W}^* \leftarrow \overrightarrow{w}^*\right] \neg\varphi \quad (4)$$

*formulates AC2 for this candidate smaller cause $\overrightarrow{X}_{(n-1)}$.*

4. *Let $\Psi$ denote the structural equations that define $M$. Let $\Psi'$ be $\Psi$ without the equations that define the variables $\overrightarrow{X}_{(n)}$ and $\overrightarrow{W}$; and let $\Psi''$ be $\Psi$ without the equations that define the variables $\overrightarrow{X}_{(n-1)}$ and $\overrightarrow{W}$. Clearly, $\Psi'' \implies \Psi'$.*
*We can turn equation 1 into a propositional formula, namely*

$$E_1 := \left(\Psi \wedge \overrightarrow{X}_{(n-1)} = \overrightarrow{x}_{(n-1)} \wedge X_n = x_n\right) \wedge \varphi. \quad (5)$$

*Similarly, equation 3 is reformulated as*

$$E_2 := \left(\Psi \wedge \overrightarrow{X}_{(n-1)} = \overrightarrow{x}_{(n-1)}\right) \wedge \varphi. \quad (6)$$

*Because equation 2 holds, we fix some $\overrightarrow{W}, \overrightarrow{w}, \overrightarrow{x}'_{(n)}$ that make it true and rewrite this equation as*

$$E_3 := \left(\Psi' \wedge \overrightarrow{X}_{(n-1)} = \overrightarrow{x}'_{(n-1)} \wedge X_n = x_n \wedge \overrightarrow{W} = \overrightarrow{w}\right) \implies \neg\varphi. \quad (7)$$

*Finally, in equation 4, we use exactly these values to also fix $\overrightarrow{W}^* = \overrightarrow{W}$, $\overrightarrow{w}^* = \overrightarrow{w}$, and $\overrightarrow{x}'^*_{(n-1)} = \overrightarrow{x}'_{(n-1)}$, and reformulate this equation as*

$$E_4 := \left(\Psi'' \wedge \overrightarrow{X}_{(n-1)} = \overrightarrow{x}'_{(n-1)} \wedge \overrightarrow{W} = \overrightarrow{w}\right) \implies \neg\varphi. \quad (8)$$

*It is then a matter of equivalence transformations to show that*

$$(\Psi'' \implies \Psi') \implies \Big((E_1 \wedge E2) \implies (E_3 \wedge E_4)\Big) \quad (9)$$

*is a tautology, which proves the lemma.*

This page intentionally left blank

# Subject Index

This page intentionally left blank

*Engineering Secure and Dependable Software Systems*
*A. Pretschner et al. (Eds.)*
*IOS Press, 2019*
*© 2019 The authors and IOS Press. All rights reserved.*

# Author Index

This page intentionally left blank

This page intentionally left blank

This page intentionally left blank