RWTH Aachen

Department of Computer Science
Technical Report

Automatic Abstraction for Bit-Vectors using Decision Procedures

Jörg Brauer

ISSN 0935–3232 · Aachener Informatik-Berichte · AIB-2013-14

RWTH Aachen · Department of Computer Science · November 2013

The publications of the Department of Computer Science of *RWTH Aachen University* are in general accessible through the World Wide Web.

http://aib.informatik.rwth-aachen.de/

Automatic Abstraction for Bit-Vectors using Decision Procedures

Von der Fakultät für Mathematik, Informatik und
Naturwissenschaften der RWTH Aachen University
zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker
Jörg Brauer
aus
Rendsburg

Berichter: Professor Dr.-Ing. Stefan Kowalewski
Professor Dr. Dr. h.c. Reinhard Wilhelm

Tag der mündlichen Prüfung: 25. September 2013

Diese Dissertation ist auf den Internetseiten der Hochschulbibliothek verfügbar.

Jörg Brauer
Verified Systems International GmbH
brauer@verified.de

Aachener Informatik Bericht AIB-2013-14

Herausgeber: Fachgruppe Informatik
RWTH Aachen University
Ahornstr. 55
52074 Aachen
GERMANY

ISSN 0935-3232

Abstract

This dissertation is concerned with abstract interpretation of programs whose semantics is defined over finite machine words. Most notably, the considered class of programs contains executable binary code, the analysis of which turns out demanding due to the complexity and the sheer number of involved operations. Challenging for correct yet precise abstract interpretation of binary code are transfer functions, which simulate the execution of any concrete operation in a program in an abstract domain. Crucially for correctness, over- and underflows need to be supported faithfully.

This dissertation argues that transfer functions and abstractions for sequences of operations over finite machine words can precisely and efficiently be generated, which contrasts with classical methods that depend on handcrafted transfer functions. To support this statement, we present an approach that eliminates the time-consuming process of manually deriving transfer functions altogether. The core of our methods are specifications of the concrete semantics of sequences of operations, which are given in propositional Boolean logic. By utilizing SAT and SMT solvers, which can determine satisfiability of Boolean formulae, we show how to automatically synthesize optimal abstractions from such semantic specifications. The practicality of our method is highlighted using abstractions generated for a variety of numerical domains that are frequently used in abstract interpretation. The abstract domains considered in this dissertation are, most notably, intervals, value sets, octagons, convex polyhedra, arithmetical congruences, affine equalities, and polynomials of bounded degree. Importantly, all presented techniques automatically handle finiteness of machine words, which manifests itself in over- and underflows.

Once the analysis of a program has terminated, an abstract interpreter often emits a warning that highlights a potential error in the analyzed program. Since abstract interpretation computes an over-approximation of the states reachable in a concrete execution, such a warning may be spurious. For this setting, we present variations of our methods, which compute complete abstractions. Then, it is possible to provide guarantees about actually reachable states, which allows us to do both, identify a warning as spurious or generate a legitimate counterexample trace.

Zusammenfassung

Diese Dissertation ist im Fachgebiet der abstrakten Interpretation angesiedelt und beschäftigt sich mit der Analyse von Programmen, deren Semantik über endlichen Registern definiert ist. Insbesondere beinhaltet diese Klasse von Programmen ausführbaren Maschinencode, dessen Analyse sich wegen der Komplexität und Anzahl der Operationen anspruchsvoll gestaltet. Eine Herausforderung für korrekte und zugleich präzise abstrakte Interpretation von Maschinencode stellen Transferfunktionen dar, welche die Ausführung aller konkreten Operationen eines Programmes in einer abstrakten Domäne simulieren. Insbesondere müssen Über- und Unterläufe von Registern modelliert werden, um die Korrektheit der Analyse zu gewährleisten.

Die in dieser Dissertation vertretene Kernthese ist, dass Transferfunktionen für Sequenzen von Operationen über endlichen Registern präzise und effizient generiert werden können. Diese These steht im Gegensatz zu klassischen Verfahren, welche den Entwurf von Transferfunktionen vornehmlich als manuellen Prozess vorsehen. Um diese These zu untermauern, präsentieren wir einen Ansatz in dem der zeitaufwendige Entwurf von Transferfunktionen entfällt. Im Mittelpunkt unseres Lösungsansatzes steht eine Spezifikation der Semantik einer Sequenz von Operationen in Form von Boolescher Logik. Mit Hilfe von Werkzeugen, welche die Erfüllbarkeit aussagenlogischer Formeln überprüfen, synthetisieren wir aus einer Spezifikation in einem automatischen Prozess Abstraktionen mit maximaler Aussagekraft. Die Praktikabilität dieses Ansatzes zeigen wir für eine Vielzahl numerischer Domänen, welche häufig in der abstrakten Interpretation eingesetzt werden, insbesondere Intervalle, Wertemengen, Oktagone, konvexe Polyeder, arithmetische Kongruenzen, affine Gleichungen und Polynome beschränkten Grades. Hierbei wird die Endlichkeit der Register, welche sich in Über- und Unterläufen manifestiert, automatisch modelliert.

Ist die Analyse eines Programms abgeschlossen, so wird von einem Analysewerkzeug unter Umständen eine Warnung ausgegeben, welche einen potentiellen Fehler im analysierten Programm anzeigt. Da abstrakte Interpretation auf Basis von Überapproximation eine Obermenge der erreichbaren Zustände eines Programms berechnet, kann solch eine Warnung unberechtigt sein. Für dieses Szenario präsentieren wir Varianten der vorgestellten Algorithmen, welche vollständige Abstraktionen generieren, so dass sich Garantien über tatsächlich erreichbare Zustände ableiten lassen. Auf dieser Basis können sowohl unberechtigte Warnungen identifiziert als auch Gegenbeispiele generiert werden, welche einen fehlerhaften Pfad anzeigen und somit die zugrundeliegende Warnung validieren.

Acknowledgements

First and foremost, I would like to thank my advisor Prof. Dr.-Ing. Stefan Kowalewski for offering me the opportunity to join his group and to write my dissertation thesis. He provided me a great degree of freedom to find my own dissertation topic, assisted me whenever I needed his advice, and supported my scientific progress in any possible way. I really appreciate the trust and responsibility I was offered despite the fact that I have only taken my first steps in academia. I also want to thank Prof. Dr. Dr. h.c. Reinhard Wilhelm for agreeing to serve on my examination committee, and also for inviting me to visit his group in Saarbrücken to both, present my work and exchange ideas.

In the past three years, the person who had the most impact on my life as a scientist has been Dr. Andy King from the University of Kent in Canterbury. Our contact started through a short email containing a few questions of mine about one of his papers, and he directly invited me over to visit him in Canterbury. His overwhelming enthusiasm, his attitude towards research, and his drive for improvement have been extremely inspiring. He is a great host and a fantastic teacher, too, and has the ability of vividly explain some of the most obscure theoretical concepts I have ever seen. It is needless to say that over the past years he has become much more than a co-author to me, and I have really enjoyed the research visits to Canterbury.

I sincerely thank all members of the Embedded Software Laboratory for a very friendly working atmosphere, which has contributed a lot to this thesis. Most importantly, I have to thank Sebastian Biallas for many interesting discussions, related or unrelated to my research. Furthermore, I thank Dr. Bastian Schlich for convincing me to join the [MC]SQUARE project in the first place. Even though he decided to leave academia several years ago, working with him for the first 1,5 years of my doctoral studies has been both, interesting and helpful.

I also want to thank Thomas Reinbacher, FH-Prof. Dr. Ing. Martin Horauer and Prof. Dr. Andreas Steininger from Vienna. Working with them on so many different topics was a great pleasure. The same holds for Prof. Kim G. Larsen whom I want to thank for giving me the opportunity to visit his group in Aalborg, Denmark for a total of three months. This stay helped me a lot broadening my understanding of verification beyond abstract interpretation. Dr. Ralf Huuck from NICTA introduced me to the world of static program analysis and was always willing to give me some advice, which was particularly helpful in the first year of my dissertation work. Similarly, Dr. Thomas Noll from the Lehrstuhl Informatik 2

in Aachen was of great help and a very pleasant and productive co-author. We had many interesting discussions on verification and he was always willing to share his experiences. I also enjoyed the interesting discussions with Dr. Axel Simon from the Technical University of Munich on the implementation of abstract interpreters and our joint work on backward reasoning. Finally, Prof. Dr. Dr. h.c. Wolfgang Thomas has put a lot of effort into the research training group *AlgoSyn*, which (together with his inspiring talks) I really appreciate.

My dissertation work was supported financially by a number of sources, including the *Deutsche Forschungsgemeinschaft* through the DFG research training group 1298 *Algorithmic Synthesis of Reactive and Discrete-Continuous Systems* and the DFG Cluster of Excellence on *Ultra-high Speed Information and Communication*, German Research Foundation grant DFG EXC 89. My work has been supported financially through a Royal Society Travel Grant, reference TG092357, and a Royal Society Project Grant, reference JP101405. I am very grateful for this support.

Furthermore, I want to thank my family, most notably my parents Karl-Ernst and Roswitha. They have always been extremely supportive, not limited to my time in Aachen. Last but certainly not least, I want to thank my beloved wife Sarah, who has been both very supporting and understanding in the last couple of years, no matter how little time I had. I suppose that living with someone whose mind circles around his dissertation topic all day and night is not always easy. Of course, my cute little dogs Pebbles and Leela shall not be forgotten either.

Jörg Brauer
Achim, November 2013

Contents

1 Introduction **1**

1.1 Abstract Interpretation of Machine Arithmetic 2
1.2 The Drive for Automatic Abstraction 2
1.3 Automatic Abstraction and Quantification 3
1.4 Automatic Abstraction using Boolean Formulae 4
1.5 Abstraction using Varieties of Domains 6
1.6 Contributions 6
1.7 Outline 7
1.8 Bibliographic Notes 8

2 Existential Quantification as Incremental SAT **9**

2.1 Prime Implicant Generation 11
2.1.1 Dual-Rail Encoding for Implicant Generation 12
2.1.2 Computing Implicants of Fixed Length 13
2.1.3 Formal Correctness 15
2.2 Anytime Quantifier Elimination 17
2.2.1 Worked Example 18
2.2.2 Formal Correctness 21
2.3 Two-Phase Quantifier Elimination 22
2.3.1 Worked Example 22
2.3.2 Formal Correctness 26
2.4 Experiments 27
2.4.1 Benchmarks 28
2.4.2 Anytime Quantifier Elimination 28
2.4.3 Two-Phase Quantifier Elimination 30
2.5 Related Work 32
2.5.1 Consensus Method and Binary Resolution 32
2.5.2 Complexity of Prime Implicant Generation 34
2.5.3 Hybrid Methods and McMillan's Method 34
2.5.4 Methods based on Prime Implicants and Cubes 35
2.5.5 Methods for Quantified Boolean Formulae 36
2.6 Discussion 36

3 Control Flow Reconstruction using Boolean Logic 37
3.1 Block-Level Abstraction . . . 40
3.1.1 Bit-Blasting Blocks . . . 40
3.1.2 Value Set Abstraction using Incremental SAT Solving . . . 41
3.1.3 Deriving Pre- and Postconditions . . . 45
3.2 Program-Level Abstraction . . . 48
3.2.1 Overview . . . 49
3.2.2 Forward Analysis with Invariant Refinement . . . 51
3.3 Experiments . . . 55
3.3.1 Benchmarks . . . 55
3.3.2 Results . . . 57
3.3.3 Comparison . . . 58
3.4 Related Work . . . 58
3.4.1 Platform-Specific Decompilation . . . 58
3.4.2 Control Flow Reconstruction by Abstract Interpretation . . . 60
3.4.3 Control Flow Reconstruction in Model Checking and Testing 61
3.4.4 Path-Sensitive Abstract Interpretation . . . 61
3.5 Discussion . . . 62

4 Automatic Abstraction of Bit-Vector Formulae 63
4.1 Separation of Modes . . . 67
4.1.1 Detecting Feasible Modes . . . 68
4.1.2 Incremental Feasibility Checks . . . 69
4.2 Symbolic Abstractions for Bit-Vectors . . . 71
4.2.1 Octagons . . . 71
4.2.2 Convex Polyhedra . . . 78
4.2.3 Non-Optimal Polyhedral Abstraction . . . 81
4.2.4 Arithmetical Congruences . . . 83
4.2.5 Affine Equalities . . . 85
4.2.6 Bounded Polynomials . . . 91
4.3 Flexible Bit-Widths by Extrapolation . . . 94
4.3.1 Templates for Extrapolation . . . 95
4.3.2 Extrapolation for Octagons . . . 96
4.3.3 Extrapolation for Affine Equalities . . . 97
4.4 Experiments . . . 97
4.4.1 Benchmarks . . . 98
4.4.2 Intervals ($\alpha^{\boldsymbol{V}}_{\mathsf{int}}$) . . . 99
4.4.3 Octagons ($\alpha^{\boldsymbol{V}}_{\mathsf{oct}}$) . . . 99
4.4.4 Convex Polyhedra ($\alpha^{\boldsymbol{V}}_{\mathsf{conv}}$) . . . 99
4.4.5 Arithmetical Congruences ($\alpha^{\boldsymbol{V}}_{\mathsf{a\text{-}cong}}$) . . . 102
4.4.6 Affine Equalities ($\alpha^{\boldsymbol{V}}_{\mathsf{aff}}$) . . . 102

4.4.7 Polynomial Equalities ($\alpha^{\boldsymbol{V}}_{\mathsf{poly}}$) . . . 106
4.4.8 Extrapolation . . . 106
4.5 Discussion . . . 106

5 Transformers for Template Constraints 111
5.1 Lifting Equalities to Template Domains . . . 113
5.1.1 Lifting Affine Equalities to Intervals . . . 113
5.1.2 Lifting Affine Equalities to Octagons . . . 116
5.1.3 Lifting Polynomial Equalities to Intervals . . . 117
5.1.4 Lifting Polynomial Equalities to Octagons . . . 120
5.2 Characterizing Linear Templates using Quantification . . . 122
5.2.1 Specifying Optimal Intervals using Quantifiers . . . 123
5.2.2 Generalization . . . 124
5.3 Interleaved Abstraction and Refinement . . . 126
5.3.1 Optimal Affine Updates on Octagons . . . 126
5.3.2 Inferring Polynomial Equalities for Octagons . . . 129
5.3.3 Optimal Affine Updates on Arithmetical Congruences . . . 130
5.4 Affine Transformers for Non-Affine Relations . . . 132
5.4.1 From Convex Polyhedra to Intervals . . . 133
5.4.2 From Convex Polyhedra to Octagons . . . 134
5.4.3 Interleaving Polyhedral Abstraction and Maximization . . . 134
5.5 Experiments . . . 135
5.5.1 Lifting and Transformation . . . 135
5.5.2 Quantification . . . 136
5.5.3 Interleaved Abstraction . . . 136
5.6 Related Work . . . 137
5.6.1 Generation of Symbolic Best Transformers . . . 138
5.6.2 Modular Arithmetic . . . 139
5.6.3 Polynomial Relations . . . 141
5.6.4 Summary-based Program Analysis . . . 142
5.7 Discussion . . . 142

6 Complete Transformers 145
6.1 Backward Analysis for Counterexamples . . . 147
6.2 Worked Example . . . 148
6.2.1 Deriving Complete Abstractions . . . 148
6.2.2 Extending Complete Abstractions . . . 148
6.2.3 Disjunctive Extensions . . . 149
6.3 Formalization . . . 150
6.3.1 Algorithm . . . 150
6.3.2 Soundness and Completeness . . . 153

6.4 Experiments . . . 154
6.4.1 Effects of Domain Combinations . . . 154
6.4.2 Complete Extrapolation . . . 155
6.5 Related Work . . . 155
6.5.1 Counterexamples in Model Checking . . . 156
6.5.2 Counterexamples in Abstract Interpretation . . . 156
6.5.3 Completeness in Abstract Interpretation . . . 157
6.6 Discussion . . . 158

7 Conclusion **159**
7.1 Discussion . . . 159
7.2 Summary . . . 161
7.3 Future Work . . . 161

1 Introduction

In model checking [8, 64], the behavior of a system or program is formally specified as a model that describes how state changes as the system progresses. All paths through the model are then exhaustively checked against its requirements, which are typically expressed in some temporal logic [97, 181]. The detailed nature of the requirements entails that the program is simulated in a fine-grained way, sometimes down to the level of individual bits [69, 208, 231]. This approach, however, may lead to state explosion since the number of states in a system is exponential in the number of system variables, the sizes of their domains, and the number of concurrent components [67]. Verification efforts based on such a fine-grained representation of the system may therefore be prohibitively expensive. Because of the complexity of this reasoning, there has been much interest in representing states and transitions of a program symbolically [51, 52], e.g., as Boolean functions [47]. This approach enables states that share some commonality to be represented without duplicating their commonality. Such methods have thus promoted the dispersal of model checking [67] and have found applications in both, hardware and software verification.

By way of comparison, the key idea in abstract interpretation [77] is to abstract away from the detailed nature of states. Then, a program analyzer operates over classes of states which are related in some sense, rather than individual states. If the number of classes is small, then all paths through the program can be examined without incurring the problems of state explosion. For example, abstract interpretation for asserting safety properties typically summarizes traces into collections of states, thereby trading the ability to distinguish traces — which is a necessity for verifying temporal logic formulae — for computational tractability. The classes of states themselves are drawn from a so-called abstract domain, which typically exhibits the structure of a (semi-) lattice. When constructed carefully, the classes of states can preserve sufficient information to prove correctness of the system. Yet, often so many details are lost when working with classes that the technique cannot infer any useful information. In these situations, the program analyzer generates a spurious warning. These so-called *false positives* are a major hindrance for the practical applicability of program analyzers [24], except those crafted for a specific application domain [83–85].

1.1 Abstract Interpretation of Machine Arithmetic

The precision of abstract interpretation frameworks [80], although sound by construction, critically depends on the expressiveness of the classes as well as the class transformers chosen to model the operations that arise in the program. Class transformers are also known as transfer functions [134]: they express how a program statement transforms a class on input into a class on output. If an input is described by a class, then the transfer function is required to simulate the execution of the instruction by computing a class which faithfully describes the output. Traditionally, transfer functions have been designed manually for each operation and each abstract domain, prior to the analysis, albeit following some well-established design principles (cp. [77, Sect. 9.2.3]). Handcrafting transfer functions, however, is difficult [115], especially when the instructions are low-level, diverse, and operate over finite machine words [186, Sect. 3]. This is because classes are themselves expressed as high-level geometric concepts such as affine [136] or polyhedral [82, 166, 224] spaces, thereby presenting a semantic gap that needs to be bridged.

1.2 The Drive for Automatic Abstraction

Recently, there has been increasing interest in computing transfer functions in a fully automatic way as part of the analysis itself [31, 32, 39, 144, 145, 197, 228–230], which is usually implemented on top of a decision procedure that computes the desired abstractions. The most descriptive transfer functions are also called *symbolic best transformers* [197], and the process of deriving them (and also non-optimal ones) is referred to as *automatic abstraction*. The high degree of automation clearly provides a way to tame the aforementioned gap between the concrete and the abstract semantics of low-level programs. Another motivation for automatic abstraction stems from the desire to reason about basic blocks, i.e., sequences of program statements, as a whole, rather than single operations. This approach was advocated by King and Søndergaard [144, 145] for linear congruences [115] in the context of verifying bit-twiddling code. Most notably, they have shown that this technique can greatly improve precision of the abstraction when there is a tight coupling between the different instructions that constitute a basic block.

To illustrate the value of block-wise abstraction compared to classical abstraction of single operations, consider a concrete operation $g : C \to C$ in a program, which is simulated using an abstract analogue $f : D \to D$. Here, C and D denote domains of concrete values and abstract descriptions, respectively. Then, f is designed to faithfully model its concrete counterpart g in the following sense: if $d \in D$ abstracts a concrete value $c \in C$, then the result of applying g to c is abstracted by applying f to d. Now suppose that a basic block is formed of n operations $g_1, \ldots, g_n$, and

each concrete operation g_i has its own abstract counterpart f_i. Suppose too that the concrete input to the sequence $c \in C$ is described by an input abstraction $d \in D$. Then, the result of applying the n concrete operations to the concrete input c is described by applying the composition of the n abstract transfer functions $f_1, \ldots, f_n$ to d, i.e., $(g_n \circ \ldots \circ g_1)(c)$ is abstracted by $(f_n \circ \ldots \circ f_1)(d)$.

However, it is important to appreciate that a more descriptive result can be obtained by deriving a single transfer function f for the basic block as a whole, designed so that f abstracts the entire sequence $g_n \circ \ldots \circ g_1$. To illustrate the gain in precision, consider a program fragment `XOR R0 R1; XOR R1 R0; XOR R0 R1` which swaps the contents of two registers `R0` and `R1` without involving a third. If the involved operations are abstracted on their own using the domain of equalities, then the outcome of the block is unknown because an exclusive-or instruction considered in isolation does not preserve equality. By way of contrast, block-wise abstraction reveals that `R0` on output equals `R1` on input, and vice versa. Of course, basic blocks are program-dependent, whereas instructions are not. Such an approach thus relies on automation rather than human intervention [31, 32, 144, 145, 167].

1.3 Automatic Abstraction and Quantification

The process of block-wise abstraction, however, critically depends on some form of quantification, and thus, on quantifier elimination algorithms [31, 167, 168]. Considering again `XOR R0 R1; XOR R1 R0; XOR R0 R1`, block-wise automatic abstraction deals with the problem of finding a relation between registers `R0` and `R1` on input and the values of the same registers on output. If the semantics of the above program fragment is expressed as a logical formula, intermediate variables within the block are required to express the relationship between the inputs and the outputs. In the above block, the value of `R0` after the first `XOR` instruction is an exemplar of such an intermediate variable. To obtain a direct relationship between the inputs and outputs, it is necessary to eliminate these variables, which are existentially quantified. The relationship is then direct in the sense that intermediate variables, which occur in the semantic specification, are removed, giving a formulation that describes the outputs of the block as a transformation of the inputs.

Monniaux [167, 169] has addressed the vexing question of automatic abstraction by focussing on linear template domains [206]. He showed that, if the concrete operations are expressed as piecewise linear functions, then it is possible to derive transformers for blocks and loops by quantifier elimination. To illustrate the role of quantification in his approach, suppose a block mutates a variable x, depending on the values of inputs y and z. To derive a transfer function for intervals, it is necessary to ascertain how the maximal value of x on exit from the block, denoted x_u, relates to the extremal values of y and z, respectively, likewise denoted y_ℓ, y_u,

z_ℓ, and z_u. The value of x_u can be specified in logic, asserting that:

1. for all values of y and z such that $y_\ell \leq y \leq y_u$ and $z_\ell \leq z \leq z_u$, the value of x is smaller than or equal to x_u, and

2. for some value of y and z such that $y_\ell \leq y \leq y_u$ and $z_\ell \leq z \leq z_u$, the variable x takes the value of x_u.

Since the "for all" can be expressed with universal quantification and the "for some" is expressed with existential quantification, these requirements are naturally formalized as a specification with alternating quantifiers. Quantifier elimination is then applied to find a direct linear relationship between the maximal value x_u of x on exit of the block and the extremal values of y and z on entry. A direct relationship between the minimal value x_ℓ of x and the extremal values of y and z can be found analogously. This construction is ingenious but no polynomial elimination algorithm is known for piecewise linear systems, or is ever likely to exist [57], and indeed, quantifier elimination remains a computational bottleneck.

1.4 Automatic Abstraction using Boolean Formulae

By way of comparison, this dissertation focusses on the problem of computing symbolic best transformers for programs whose semantics is specified over finite bit-vectors, which is akin to reasoning about assembly or binary code, rather than piecewise linear systems. We suggest to express the semantics of programs in the computational domain of propositional Boolean formulae — an idea that is familiar in model checking [69] where it is colloquially referred to as bit-blasting — and directly benefit from impressive progress on automatic decision procedures such as SAT or SMT solvers. Since bit-vector formulae are more expressive than piecewise linear formulae, one would expect automatic abstraction to be at least as difficult; or even harder, since these formulae are discrete. From a theoretical point of view, this is a reasonable expectation. Yet, careful engineering and elegant ideas have advanced SAT and SMT solvers to the point they can rapidly decide satisfiability of structured problems involving thousands of variables and clauses [50].

In essence, it is our thesis that SAT and SMT solving, as an elementary mechanism for reasoning about correctness of programs whose semantics is expressed over finite machine words, has become a practical and scalable proposition to implement and compute abstract interpretations. One advantage of this approach is that it is amenable to instructions whose semantics is presented as Boolean formulae. This advantage dovetails with the rise in popularity of SAT-based (bounded or unbounded) model checking since propositional encodings are readily available for instructions [39, 69, 111, 186]. Encodings for other, non-standard operations, can easily be derived from commonly known templates [19, 39, 43, 214]. When

representing the semantics of real-world programs using piecewise linear systems, it is often unclear how to express some instructions (such as bitwise exclusive-or), whereas a specification in Boolean logic can be derived straightforwardly.

By reformulating the approach described by Monniaux [167, 169] in Boolean logic, we obtain a collection of different methods that avoid the computational problems associated with eliminating universally quantified variables from linear systems. Whereas automatic abstraction for linear systems relies on complicated elimination algorithms, this is different for Boolean formulae in conjunctive normal form (CNF). To illustrate, consider computing $\forall x : \varphi$, where $\varphi = (x \vee \neg y) \wedge (\neg x \vee y \vee \neg z)$. By expanding the formula to eliminate x, we obtain:

$$\forall x : \varphi \quad = \quad \varphi[x \mapsto 0] \wedge \varphi[x \mapsto 1] \quad = \quad (\neg y) \wedge (y \vee \neg z)$$

Observe that $\forall x : \varphi$ can likewise be obtained directly from φ by removing all literals that involve x from φ. This is no coincidence, and indeed the approach can be applied to any formula in CNF not containing a vacuous clause [148]. The simplicity of universal quantifier elimination, however, contrasts with the difficulty of existential elimination. Yet, as one contribution of this dissertation, we will see that it is possible to efficiently implement this operation using algorithms [33, 42] which are based on incremental SAT solving [242].

Crucially for performance, we also provide methods that avoid quantifier elimination altogether. Then, a solver is repeatedly invoked to find models of a Boolean formula. This approach contrasts with that of deriving a formula that specifies abstract relations directly on the bit-level. As an example, consider an octagonal inequality $x + y \leq d$ [166]. The constant $d \in \mathbb{Z}$ is characterized as $d = \min\{c \in \mathbb{Z} \mid \forall x : \forall y : P(x, y) \Rightarrow (x + y \leq c)\}$ where $P(x, y)$ is a predicate constraining x and y. Further, given a machine with word-length w, the maximal value of a register in an unsigned interpretation is $2^w - 1$. We can thus derive a constraint $0 \leq d \leq 2 \cdot (2^w - 1)$ for d, which can be expressed disjunctively as:

$$(0 \leq d \leq 2^w - 1) \quad \vee \quad (2^w \leq d \leq 2 \cdot (2^w - 1))$$

To determine which disjunct characterizes d, it is sufficient to test the formula $\exists x : \exists y : P(x, y) \wedge (x + y \geq 2^w)$ for *satisfiability*. If satisfiable, then $2^w \leq d \leq 2 \cdot (2^w - 1)$ is entailed by d, and $0 \leq d \leq 2^w - 1$ otherwise. We proceed by decomposing the new characterization into a disjunction and repeating this step w times to give d exactly. Hence, reformulating the problem as a sequence of queries to a SAT solver finesses the problem of quantifier elimination. An attractive property of this approach is that the method will profit directly from any future progress made on off-the-shelf SAT and SMT solvers [94, 159], which are readily (and freely) available.

1.5 Abstraction using Varieties of Domains

A further compelling attribute of our approach is that the interdependence between the low-level description of the concrete semantics of a program and its high-level geometric abstraction is dissolved. Given a logical formulation of the concrete semantics of program statements, we present a framework which supports the generation of transformers for virtually any abstract domain, solely limited to those of finite height, and derive dedicated procedures that efficiently compute abstractions of bit-vector relations for a set of different abstract domains. The aforementioned limitation to finite domains indeed is an insignificant one for reasoning about bit-vector programs since state can always be represented finitely (the termination problem, e.g., is decidable and countable [75, App. B]). Using this method, a large number of abstract interpretations can be derived directly from the concrete semantics of the program without human intervention; in this dissertation, we present methods for intervals, value sets, arithmetical congruences, octagons, convex polyhedra, affine equalities, and polynomial equalities of bounded degree.

Support for a set of abstract domains is indeed a key feature of our approach, as state-of-the-art abstract interpreters such as Astrée rely on combinations of abstract domains to keep the number of spurious alarms at a minimum [83–85]. This is because the domains can capture different behaviors, and the framework of abstract interpretation provides a way to combine the information discovered by each abstract domain using a construction known as the reduced product [77, 86], or variations thereof. Automatic abstraction on top of Boolean logic thus provides a suitable technology not only to tame the difficulty of handcrafting transfer functions, but also reduces the workload for supporting different abstract domains.

1.6 Contributions

The key contributions of this thesis are:

- We present two algorithms for existential quantification on propositional Boolean formulae in CNF using incremental SAT solving. The first algorithm is designed to be *anytime* — it can be interrupted prematurely — whereas the second one dismisses this property in favor of efficiency.
- We show that quantifier elimination can form part of SAT-based value set analysis, which we apply to automatically compute invariants from unstructured binary code to recover a sound, yet precise, control flow graph.
- We adapt the formulation of automatic abstraction based on Monniaux [167, 169] to the setting of quantified Boolean formulae, thereby benefiting from a tractable form of universal quantification for CNF [148, Chap. 9.2.3].

- We further provide algorithms which compute the least sound abstraction of a propositional Boolean formula in different abstract domains using incremental SAT solving. Specifically, we support value sets, intervals, octagons, convex polyhedra, arithmetical congruences, affine equalities, and bounded polynomials. For efficiency, these algorithms exploit the structure of the underlying abstract domains, and thus outperform general-purpose ones [197].

- We study different techniques to derive transformers from the aforementioned abstractions. These transfer functions then express, e.g., how an octagon on input into is transformed into an octagon on output using a polynomial map.

- We provide a technique for extrapolation, based on the observation that programs defined using the same instructions but over differing bit-widths frequently exhibit a high degree of similarity. Yet, deriving abstractions for bit-vectors of width, say, 64 is challenging performance-wise. We therefore show how to extract small-scale abstractions from small bit-vectors — e.g., of width 8 — and then lift these relations to wider bit-vectors.

- We propose a technique to find legitimate traces that reveal actual defects using backward analysis. To do so, we present an algorithm that converges onto a complete abstraction from below. Resting backward analysis on completeness then allows us to compute weakest preconditions, which is required to classify a counterexample trace as either spurious or legitimate.

In summary, this thesis presents techniques that automatically generate abstractions of programs defined over finite machine words. Given a logical characterization of the basic blocks that constitute the program, abstractions and transformers are computed, rather than designed manually. This entails that a program analyzer based on our methods needs to provide (1) logical characterizations of the program statements, (2) an algorithm that provides fixed-point iteration, and (3) implementations of standard lattice operations such as join in the abstract domains used. The cumbersome and difficult part of implementing and evaluating the concrete operations of a program in an abstract setting is avoided altogether.

1.7 Outline

Expositionally, our contributions are laid out as follows. First, Chap. 2 presents two algorithms for SAT-based existential quantifier elimination. Then, Chap. 3 presents a method that uses our quantification technique to compute interval and value set abstractions from executable code. These abstractions are then used to reconstruct jump targets from the binary, thereby providing a sound approximation of the control flow graph. The following Chap. 4 focusses on abstractions for the

domains of octagons, convex polyhedra, arithmetical congruences, affine equalities, and polynomial equalities of bounded degree. For each abstract domain, a dedicated abstraction procedure is provided. From the abstractions, we derive transfer functions that express how an abstract state on input to a block is mapped to an abstract state on output, which is detailed in Chap. 5. Whereas Chap. 4 and Chap. 5 focus on sound abstractions, Chap. 6 targets the problem of computing complete abstractions so as to infer definitive counterexample traces. The aforementioned chapters each contain experimental results and comparisons to related work. Finally, this dissertation concludes with a discussion in Chap. 7.

1.8 Bibliographic Notes

This dissertation is, at least to some extent, based on work that we have described in refereed publications. Lines of argumentation similar to those used in this chapter have been used to motivate our work in [31–35]. Prime implicant generation, which is described in Chap. 2.1, is part of algorithms presented in two conference papers [33, 42]. Following, Chap. 2.2 and Chap. 2.3 are based on these two publications. Likewise, Chap. 3, which presents a SAT-based value set analysis for control flow reconstruction, is based on two papers [41, 187]. The abstraction procedure for octagons in Chap. 4.2.1 was first presented in [32], whereas affine abstraction in Chap. 4.2.5 was introduced in [31]. The characterization of transformers as quantified Boolean formulae in Chap. 5.2 has first been described in [31]. However, the technique presented in [31] used resolution-based quantifier elimination, rather than the SAT-based techniques employed in this thesis. An extended article [34], which generalizes the paper [32], is currently under review. Chapter 6, which describes the generation of complete abstractions, is inspired by a conference publication, too [35]. However, [35] relies on state representations and transformers in Boolean logic, whereas Chap. 6 presents novel, complete abstractions for domains such as affine equalities. All techniques for reasoning about relations between program variables are based on relational encodings for bit-vector programs, which are heavily inspired by the AVR and Intel MCS-51 microcontroller instruction sets. We have described similar encodings in previous works [31, 32, 34, 35, 39, 43, 187].

During the course of our doctoral studies, we have worked in different areas which are not directly related to this dissertation. Most notably, we have worked on state-space reductions for explicit-state model checking based on static analysis [25, 118, 119, 188, 189, 191, 210, 211], on verification techniques for programmable logic controllers [26–28, 209], on control-structure refinement using abstract interpretation [38], on pointer analysis and low-level memory models [22, 36], on stack analysis [37], and on runtime verification of microcontroller code [190, 192–194].

2 Existential Quantification as Incremental SAT

The problem of eliminating variables from Boolean formulae using existential quantification is ubiquitous in formal verification [2, 29, 75, 103, 104, 152, 164]. The benefit of novel, improved algorithms which solve this problem is thus not limited to our indicated applications in automatic abstraction of bit-vector programs. As a concrete scenario, consider predicate abstraction [107], from which we take an example that we develop in what follows. In predicate abstraction, a finite set of predicates is used to express properties of — and relationships between — program variables at different points in the program. An abstract state of the program in this domain can then be described by a cube — a conjunction of literals — over the set of predicate symbols, and likewise a set of states as a disjunction of cubes. As in symbolic model checking [51, Sect. 2], existential quantifier elimination then arises during the computation of successor states. Adapting an example from [152], suppose predicates $X = \{x_1, \ldots, x_6\}$ and $Y = \{y_1, \ldots, y_6\}$ express state at two consecutive program points. Further, suppose the transition relation between the states at these two program points is expressed as a Boolean function over $X \cup Y$:

$$\mu \quad = \quad \begin{cases} \neg(x_2 \wedge y_2) \wedge \neg(y_1 \wedge y_2) \wedge ((x_4 \wedge x_6) \Rightarrow y_1) & \wedge \\ (x_3 \Leftrightarrow y_4) \wedge (x_4 \Leftrightarrow y_3) \wedge (x_5 \Leftrightarrow y_6) \wedge (x_6 \Leftrightarrow y_5) \end{cases}$$

If the set of states at one program point is described by the formula

$$\xi \quad = \quad \begin{cases} (x_1 \wedge \neg x_2 \wedge x_3 \wedge \neg x_4 \wedge x_5 \wedge \neg x_6) & \vee \\ (x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge x_4 \wedge \neg x_5 \wedge x_6) \end{cases}$$

then the state at the next program point is given as $\exists X : \xi \wedge \mu$. Existential quantifier elimination, which is in this context also referred to as image computation, then amounts to removing from $\xi \wedge \mu$ all information pertaining to the variables in X. The operation thus yields a formula that ranges only over the predicates Y.

Quantifier Elimination using Model Enumeration This chapter is concerned with techniques for computing a quantifier-free formula $\exists X : \varphi$ in CNF using incremental SAT solving, where X and Y are sets of propositional variables and φ is itself presented in CNF. The quadratic nature of each transformation step renders the

application of traditional techniques such as binary resolution impractical when X is large compared to $\mathsf{vars}(\varphi)$ [148, Chap. 9.2.3]. Sometimes, binary decision diagrams (BDDs) are used to compute $\exists X : \varphi$ [47, 51, 102]. Yet, SAT-based techniques are favored when the set of of variables $Y = \mathsf{vars}(\varphi) \setminus X$ is small compared to $\mathsf{vars}(\varphi)$ [50, 152]. Such techniques invoke a SAT solver to find a model of φ, and then extract a cube

$$c_1 \quad = \quad (\bigwedge\nolimits_{y_1 \in Y_1} y_1) \quad \wedge \quad (\bigwedge\nolimits_{y_2 \in Y_2} \neg y_2)$$

for which $\varphi \wedge c_1$ is satisfiable and Y_1 and Y_2 partition Y, i.e., $Y_1 \cap Y_2 = \emptyset$ and $Y_1 \cup Y_2 = Y$. Then, c_1 entails $\exists X : \varphi$, formally $c_1 \models \exists X : \varphi$. The blocking clause $\neg c_1$ is then added to φ to give $\varphi \wedge \neg c_1$ and the process is repeated to enumerate all such cubes $c_1, \ldots, c_n$. During enumeration, the cubes $c_1, \ldots, c_n$ are typically stored in a BDD which converges onto $\exists X : \varphi$ from below. Each intermediate BDD clearly under-approximates $\exists X : \varphi$. A CNF representation of $\bigvee_{i=1}^{n} c_i$ can then be extracted from the BDD, e.g., by following all (conjunctive) paths $d_1, \ldots d_m$ in the BDD which lead to the false leaf and negating the disjunction $\bigvee_{i=1}^{m} d_i$ to give $\bigwedge_{i=1}^{m} \neg d_i$ in CNF. In his seminal work on SAT-based unbounded symbolic model checking, McMillan [164, Sect. 1] critiqued this approach by pointing out that:

> "*CNF and SAT-based quantifier elimination can be exponentially more efficient than [..] BDDs in cases where the resulting fixed points have compact representations in CNF, but not as BDDs.*"

BDDs have been used to store the cubes as it is believed that they offer a time- and space-efficient data structure for storing the image, i.e., the quantifier-free formula. Further, contemporary BDD packages such as Cudd [226] or Buddy [72] provide built-in support to convert a BDD into CNF. BDDs can thus straightforwardly be combined with SAT-based approaches to quantifier elimination. However, this does not preclude computing CNF directly, especially if the size of the CNF is smaller than that of the BDD [164, Sect. 6].

Contributions Our contribution can be considered a response to the agenda set by McMillan. It consists of two novel methods for computing the image as a compact CNF formula. The novelty of our algorithms, compared to those of McMillan and others, is that they do neither require modifications to a solver nor integration with BDDs. They can thus compactly be implemented on top of existing SAT engines using a few hundred lines of program code only. In both approaches, the quantifier elimination problem is reduced to that of finding cubes of minimal size, each of which entails $\exists X : \varphi$. We show how this problem can be encoded as a SAT instance if properly combined with sorting networks so as to obtain cardinality constraints (see Chap. 2.1). This encoding is the key ingredient of both algorithms. Yet, the algorithms presented herein differ in the following sense:

- The first algorithm (see Chap. 2.2) is designed to incrementally produce a sequence of formulae that converges onto $\exists X : \varphi$ from above. Hence, the algorithm possesses the attractive *anytime* property, i.e., it "*[..] can be terminated at any time and will return some answer [..] in some well-behaved manner as a function of time*" [92, Sect. 3]. Such algorithms can thus be preempted prematurely without compromising soundness [23].

- The second algorithm (see Chap. 2.3) decomposes the elimination problem into two phases. First, prime implicant enumeration is applied to compute a DNF representation of $\exists X : \varphi$, followed by CNF conversion using the same technique. The former phase is non-interruptible (the latter phase still is), but the algorithm is more efficient.

We show that our formulation finesses the need for a complicated quantifier elimination procedure such as the DPLL-style algorithm by McMillan [164, Sect. 2] or model enumeration using heuristics by Lahiri et al. [152, Sect. 3.2]. Furthermore, with a BDD-based approach, the sizes of the BDD itself and of the resulting CNF formula (and thus the runtime needed to compute it) are very sensitive to the variable ordering (cp. [49, 70]). This holds true even when dynamic reordering is applied. By way of comparison, the algorithms described herein actually produce a compact CNF representation without the application of heuristics, thereby challenging the belief that BDDs are necessary for existential quantifier elimination.

Structure of the Presentation Each of the key ingredients of this chapter, namely, prime implicant generation as well as the two different quantifier elimination algorithms, is presented in its own section. In each section, the algorithm is presented by means of a worked example, followed by a discussion of formal correctness arguments. Experimental results are then presented in Chap. 2.4. Finally, the chapter concludes with a presentation of related work in Chap. 2.5 and a discussion in Chap. 2.6.

2.1 Prime Implicant Generation

An implicant of a propositional Boolean formula φ is a cube c such that $c \models \varphi$ and $\mathsf{vars}(c) \subseteq \mathsf{vars}(\varphi)$, i.e., every model of c is also a model of φ. A prime implicant of φ, in turn, is an irreducible cube that entails φ. Therefore, shortest prime implicants deliver the most descriptive under-approximations of φ among all implicants. The key idea behind both techniques for quantifier elimination is thus to compute shortest prime implicants of a propositional Boolean formula $\exists X : \varphi$ so as to rapidly converge onto the set of solutions of $\exists X : \varphi$. To do so, it is necessary to draw the literals in each implicant c only from variables in Y. This chapter presents a dual-rail transformation [48] of the problem which serves this task. Then, the transformed

formula is passed to a SAT solver and the resulting model is turned into a prime implicant of $\exists X : \varphi$. The encoding is further paired with sorting networks [146] to enforce cardinality constraints [95] on the sizes of the implicants. The force of this construction is that it allows us to derive shortest prime implicants of $\exists X : \varphi$ automatically using SAT solving; it is thus fundamental for the algorithms presented in Chap. 2.2 and Chap. 2.3. We build towards this technique using $\varphi = (\xi \wedge \nu)$ from the introduction and demonstrate how to compute prime implicants of $\exists X : \varphi$.

2.1.1 Dual-Rail Encoding for Implicant Generation

To compute an implicant of $\exists X : \varphi$ using SAT solving, it is first necessary to convert φ into CNF, for which we introduce a set of fresh Tseitin variables T [178, 234]. The fresh variables are themselves existentially quantified. The according transformation yields a formula ψ which is equisatisfiable to φ. Then, $\mathsf{vars}(\psi) = X \cup Y \cup T$ and suppose that $Y = \{y_1, \ldots, y_k\}$. We introduce two disjoints sets

$$Y^+ = \{y_1^+, \ldots, y_k^+\} \qquad Y^{-1} = \{y_1^-, \ldots, y_k^-\}$$

of fresh variables and replace each occurrence of the positive literal y_i in ψ with y_i^+. Likewise, we replace each occurrence of $\neg y_i$ in ψ with y_i^-. To ensure that y_i^+ and y_i^- cannot hold simultaneously, the transformed formula is further augmented with:

$$\textstyle\bigwedge_{i=1}^{k} (\neg y_i^+ \vee \neg y_i^-)$$

Let $\tau(\psi)$ denote this syntactic transformation, which yields a formula in CNF that is defined over propositional variables $V = X \cup Y^+ \cup Y^- \cup T$. The intuitive idea behind this transformation is to construct a formula that has the same models as ψ but does not force each variable $y_i \in Y$ to be assigned either 0 or 1. Instead, a model of $\tau(\psi)$ may contain assignments of 0 for y_i^+ as well as for y_i^-. Such a case entails that the model holds for either value of y_i, i.e., its validity does not depend on y_i. Passing $\tau(\psi)$ to a SAT solver yields a model $\mathbf{m}_1 : V \to \mathbb{B}$, such as:

$$\mathbf{m}_1 \;=\; \left\{ \begin{array}{llllll} x_1 \mapsto 1, & x_2 \mapsto 0, & x_3 \mapsto 1, & x_4 \mapsto 0, & x_5 \mapsto 1, & x_6 \mapsto 0 \\ y_1^+ \mapsto 0, & y_2^+ \mapsto 0, & y_3^+ \mapsto 0, & y_4^+ \mapsto 1, & y_5^+ \mapsto 0, & y_6^+ \mapsto 1 \\ y_1^- \mapsto 1, & y_2^- \mapsto 1, & y_3^- \mapsto 1, & y_4^- \mapsto 0, & y_5^- \mapsto 1, & y_6^- \mapsto 0 \end{array} \right\}$$

Note that the Tseitin variables T have been omitted for the purpose of presentation. The same model $\mathbf{m}_1$ can be represented as a subset of V, namely:

$$\{v \in V \mid \mathbf{m}_1(v) = 1\}$$

Henceforth, we shall use both representations interchangeably. Then, the variables in $\mathbf{m}_1 \cap (Y^+ \cup Y^-)$ define a conjunction of literals $\rho_Y(\mathbf{m}_1)$ over the variables Y as:

$$\begin{array}{rcl} \rho_Y(\mathbf{m}_1) & = & \big(\bigwedge\{y_i \mid y_i^+ \in \mathbf{m}_1 \cap Y^+\}\big) \wedge \big(\bigwedge\{\neg y_i \mid y_i^- \in \mathbf{m}_1 \cap Y^-\}\big) \\ & = & (\neg y_1 \wedge \neg y_2 \wedge \neg y_3 \wedge y_4 \wedge \neg y_5 \wedge y_6) \end{array}$$

The cube $\rho_Y(\mathbf{m}_1)$ entails $\exists X : \varphi$ as well as $\exists X : \exists T : \psi$. However, please observe that $\rho_Y(\mathbf{m}_1)$ is not a prime implicant of $\exists X : \varphi$ because the SAT solver could also have produced a model $\mathbf{m}_2$ defined as:

$$\mathbf{m}_2 \quad = \quad \left\{ \begin{array}{llllll} x_1 \mapsto 1, & x_2 \mapsto 0, & x_3 \mapsto 1, & x_4 \mapsto 0, & x_5 \mapsto 1, & x_6 \mapsto 0 \\ y_1^+ \mapsto 0, & y_2^+ \mapsto 1, & y_3^+ \mapsto 0, & y_4^+ \mapsto 1, & y_5^+ \mapsto 0, & y_6^+ \mapsto 1 \\ y_1^- \mapsto 1, & y_2^- \mapsto 0, & y_3^- \mapsto 1, & y_4^- \mapsto 0, & y_5^- \mapsto 1, & y_6^- \mapsto 0 \end{array} \right\}$$

Then, $\rho_Y(\mathbf{m}_2) = (\neg y_1 \wedge y_2 \wedge \neg y_3 \wedge y_4 \wedge \neg y_5 \wedge y_6)$ and $\rho_Y(\mathbf{m}_2) \models \exists X : \varphi$. From this we can also deduce $\rho_Y(\mathbf{m_1}) \vee \rho_Y(\mathbf{m}_2) \models \exists X : \varphi$, and simplification gives:

$$\rho_Y(\mathbf{m}_1) \vee \rho_Y(\mathbf{m}_2) \quad = \quad (\neg y_1 \wedge \neg y_3 \wedge y_4 \wedge \neg y_5 \wedge y_6)$$

We clearly have $(\neg y_1 \wedge \neg y_3 \wedge y_4 \wedge \neg y_5 \wedge y_6) \models \exists X : \varphi$. This entails that $\rho_Y(\mathbf{m}_1)$ and $\rho_Y(\mathbf{m}_2)$ are both reducible implicants of $\exists X : \varphi$, since $(\neg y_1 \wedge \neg y_3 \wedge y_4 \wedge \neg y_5 \wedge y_6)$ provides a more descriptive under-approximation of $\exists X : \varphi$ than $\rho_Y(\mathbf{m}_1)$ or $\rho_Y(\mathbf{m}_2)$ in separation. In the following section, we will therefore refine this encoding and demonstrate how to systematically infer shortest prime implicants.

2.1.2 Computing Implicants of Fixed Length

To derive shortest prime implicants, we turn to sorting networks [146]. Examples of sorting networks for 3 and 4 bits are given in Fig. 2.1. The 3-bit sorter on the left-hand side of Fig. 2.1 has 3 input bits on the left and 3 output bits on the right. It also contains 3 comparison operations, indicated with vertical bars, which compare and if necessary swap bits. A comparator assigns its outgoing upper bit to the maximum of its two incoming bits and its outgoing lower bit to the minimum. Since maximum and minimum in this context correspond to the logical symbols $\vee$ and $\wedge$, a comparator with incoming bits i_1 and i_2 as well as outgoing bits o_1 and o_2 can be encoded propositionally as the formula:

$$(o_1 \leftrightarrow i_1 \vee i_2) \wedge (o_2 \leftrightarrow i_1 \wedge i_2)$$

The value of a sorting network is that it can be applied to express the sum of k propositional variables in no more than $12k(\lceil \log_2(k) \rceil + 1)$ ternary clauses where the sum is represented in a unary fashion [95]. The 3-bit sorter in Fig. 2.1, for instance, over inputs $I = \{i_1, i_2, i_3\}$, outputs $O = \{o_1, o_2, o_3\}$ and internal auxiliary bits $A = \{a_1, a_2, a_3\}$, can be encoded propositionally as:

$$\sigma_3 \quad = \quad \left\{ \begin{array}{lllll} (a_1 \leftrightarrow i_1 \vee i_2) & \wedge & (a_2 \leftrightarrow i_1 \wedge i_2) & \wedge \\ (a_3 \leftrightarrow a_2 \vee i_3) & \wedge & (o_3 \leftrightarrow a_3 \wedge i_3) & \wedge \\ (o_1 \leftrightarrow a_1 \vee a_3) & \wedge & (o_2 \leftrightarrow a_1 \wedge a_3) & \end{array} \right.$$

Moreover, by instantiating the output bits to fixed unary value, a cardinality constraint can be obtained. For example, by constraining the output bits of the 4-bit sorter to 1100, a cardinality constraint is derived which ensures that exactly 2 of the 4 input bits to the sorter are set. Constraining the output bits to 1110 would ensure that exactly 3 input bits are set. Such cardinality constraints can be imposed in conjunction with the formula $\tau(\psi)$ in order to enforce the discovery of the shortest implicants first, and thereby prevent redundant implicants to be found. Returning to the example discussed in the previous section, we observe that the aforementioned redundancy by implicants $\rho_Y(\mathbf{m}_1)$ and $\rho_Y(\mathbf{m}_2)$ would not have occurred, had the SAT solver first found the following model:

$$\mathbf{m}_3 \;=\; \left\{ \begin{array}{llllll} x_1 \mapsto 1, & x_2 \mapsto 0, & x_3 \mapsto 1, & x_4 \mapsto 0, & x_5 \mapsto 1, & x_6 \mapsto 0 \\ y_1^+ \mapsto 0, & y_2^+ \mapsto 0, & y_3^+ \mapsto 0, & y_4^+ \mapsto 1, & y_5^+ \mapsto 0, & y_6^+ \mapsto 1 \\ y_1^- \mapsto 1, & y_2^- \mapsto 0, & y_3^- \mapsto 1, & y_4^- \mapsto 0, & y_5^- \mapsto 1, & y_6^- \mapsto 0 \end{array} \right\}$$

To guarantee that shortest prime implicants are found first, we introduce a set $Y^{\pm} = \{y_1^{\pm}, \ldots, y_k^{\pm}\}$ of fresh variables. These variables serve as inputs to a k-bit sorting network. Each variable $y_i^{\pm} \in Y^{\pm}$ indicates whether y_i^+ or y_i^- appears in the implicant; if so, $y^{\pm}$ evaluates to 1, and to 0 otherwise. To connect the y_i^+ and y_i^- variables to the respective $y_i^{\pm}$, a conjunction $\bigwedge_{i=1}^{k} \theta_i$ is introduced, where:

$$\begin{array}{rcl} \theta_i & = & y_i^{\pm} \Leftrightarrow (y_i^+ \vee y_i^-) \\ & = & (\neg y_i^{\pm} \vee y_i^+ \vee y_i^-) \wedge (y_i^{\pm} \vee \neg y_i^+) \wedge (y_i^{\pm} \vee \neg y_i^-) \end{array}$$

Given a k-bit sorter σ_k with output variables $\{o_1, \ldots, o_k\}$, a formula whose models describe implicants of $\exists X : \varphi$ of length ℓ with $1 \leq \ell \leq k$ is obtained by augmenting $\tau(\psi)$ with a constrained sorting network as follows:

$$\tau_\ell(\psi) \;=\; \tau(\psi) \wedge \sigma_k \wedge \left(\bigwedge_{i=1}^{k} \theta_i\right) \wedge \left(\bigwedge_{i=1}^{\ell} o_i\right) \wedge \left(\bigwedge_{i=\ell+1}^{k} \neg o_i\right)$$

Since $\tau_\ell(\psi)$ is unsatisfiable for $l \in \{1, \ldots, 4\}$, we deduce that $\exists X : \varphi$ does not possess implicants shorter than 5. Testing $\tau_5(\psi)$ for satisfiability yields the model $\mathbf{m}_3$ as above, which defines:

$$\rho_Y(\mathbf{m}_3) \;=\; (\neg y_1 \wedge \neg y_3 \wedge y_4 \wedge \neg y_5 \wedge y_6)$$

Adding a blocking clause $\neg\rho_Y(\mathbf{m}_3)$ to φ suppresses both, $\mathbf{m}_1$ and $\mathbf{m}_2$. Note that although the clauses $(\bigwedge_{i=1}^{\ell} o_i) \wedge (\bigwedge_{i=\ell+1}^{k} \neg o_i)$ must be rescinded once all the implicants of length ℓ have been found, this sub-formula is itself a cube. The force of this is that SAT solvers support assumptions which are cubes. The assumption is added to the instance, thereby binding some variables, but these bindings are discarded once a model is found, in readiness for the next call to the solver. Conveniently,

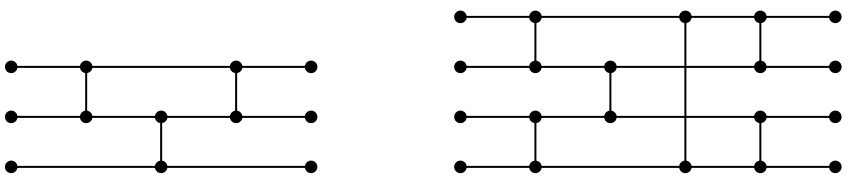


Figure 2.1: Sorting networks for 3 and 4 bits; vertical bars indicate comparators

this lightweight version of incremental SAT is sufficient to support the above algorithm. To summarize, we have thus far introduced a transformation map τ for a dual-rail encoding which, if paired with sorting networks and respective cardinality constraints, can be used to derive shortest prime implicants of $\exists X : \varphi$.

2.1.3 Formal Correctness

To prove the formula transformations introduced so far correct, let Bool_V denote the class of propositional formulae over the set of variables V. Suppose that two disjoint sets X and Y partition V, i.e., $X \cap Y = \emptyset$ and $X \cup Y = V$. We shall now consider the problem of computing an implicant of $\exists X : \varphi$ where the formula $\varphi \in \mathsf{Bool}_V$ is presented in CNF. The transformation is formalized as a map τ on the set of literals $\mathsf{Lit}_V = \{v, \neg v \mid v \in V\}$. This map is, in turn, defined in terms of sets of propositional variables $Y^+ = \{y^+ \mid y \in Y\}$ and $Y^- = \{y^- \mid y \in Y\}$ for which we assume that $Y^+ \cap Y^- = \emptyset$ and $(Y^+ \cup Y^-) \cap V = \emptyset$.

Definition 2.1. *The literal transformation map* $\tau : \mathsf{Lit}_V \to \mathsf{Lit}_{Y^+ \cup Y^- \cup X}$ *and its inverse* $\tau^{-1} : \mathsf{Lit}_{Y^+ \cup Y^- \cup X} \to \mathsf{Lit}_V$ *are defined as follows:*

$$\tau(l) = \begin{cases} y^+ & : l = y \text{ and } y \in Y \\ y^- & : l = \neg y \text{ and } y \in Y \\ l & : \text{otherwise} \end{cases} \qquad \tau^{-1}(l) = \begin{cases} y & : l = y^+ \text{ and } y \in Y \\ \neg y & : l = y^- \text{ and } y \in Y \\ l & : \text{otherwise} \end{cases}$$

A clause is considered to be a set of literals to simplify the lifting of the literal transformation map to clauses. Thus, if a clause is merely a set $C \subseteq \mathsf{Lit}_V$ then $\tau(C) = \{\tau(l) \mid l \in C\}$. Likewise, τ can be lifted to a formula $\varphi \subseteq \wp(\mathsf{Lit}_V)$. Given a set $W \subseteq \mathsf{Lit}_V$ of literals, we sometimes use $\bigwedge W$ as a shortcut for the formula $\bigwedge_{w \in W} w$. Likewise, we sometimes use $\bigvee W$ for $\bigvee_{w \in W} w$.

Definition 2.2. *Let* $F \subseteq \wp(\mathsf{Lit}_V)$ *and let* $\varphi = \bigwedge\{\bigvee C \mid C \in F\}$. *The formula transformation map* $\tau : \wp(\mathsf{Lit}_V) \to \wp(\mathsf{Lit}_{X,Y})$ *is defined:*

$$\tau(\varphi) \quad = \quad \bigwedge\{\bigvee \tau(C) \mid C \in F\} \wedge \bigwedge\{\neg y^+ \vee \neg y^- \mid y \in Y\}$$

An implicant of φ is a cube c such that $c \models \varphi$. Our interest is in cubes which are non-trivial, that is, cubes that do not contain opposing literals. The classes of cubes over literals from V and $X \cup Y^+ \cup Y^-$ are given through the following definition.

Definition 2.3. *The classes of non-trivial cubes are defined below:*

$$\begin{aligned} \mathsf{Cube}_V &= \left\{ C \subseteq \mathsf{Lit}_V \;\middle|\; \forall v \in V : \{v, \neg v\} \not\subseteq C \right\} \\ \mathsf{Cube}_{X,Y} &= \left\{ C \cup C' \;\middle|\; \begin{array}{lll} C \in \mathsf{Cube}_X & \wedge & C' \subseteq Y^+ \cup Y^- \\ \forall y \in Y : \{y^+, y^-\} \cap C' \neq \emptyset & \wedge & \{y^+, y^-\} \not\subseteq C' \end{array} \right\} \end{aligned}$$

The literal transformation map τ is lifted to cubes by likewise considering these to be sets of (implicitly conjoined) literals. The transformation relates cubes with literals drawn from Lit_V to cubes with literals drawn from $Y^+ \cup Y^- \cup \mathsf{Lit}_X$. The above definitions allow us to state equivalence between cubes over the original formula φ and those obtained by applying the formula translation $\tau(\varphi)$.

Proposition 2.1 (Equivalence)**.** *Let $F \subseteq \wp(\mathsf{Lit}_V)$ and let $\varphi = \bigwedge\{\bigvee C \mid C \in F\}$ denote a formula in CNF. Further, put $\varphi' = \bigwedge\{\bigvee \tau(C) \mid C \in F\}$. Then:*

1. *If $D \in Cube_V$ and $(\bigwedge D) \models \varphi$ then $(\bigwedge \tau(D)) \models \varphi'$.*
2. *If $D' \in Cube_{X,Y}$ and $(\bigwedge D') \wedge \varphi'$ is satisfiable then $(\bigwedge \tau^{-1}(D') \models \varphi$.*

Proof of Proposition 2.1. We prove correctness of the two statements separately.

1. Let $C \in F$. Since we have $(\bigwedge D) \models \varphi$, it follows that $(\bigwedge D) \models (\bigvee C)$. We then have to consider the following four different cases:
   - Suppose $x \in D \cap C$ and $x \in Y$. Then $x^+ \in \tau(C) \cap \tau(D)$.
   - Suppose $\neg x \in D \cap C$ and $x \in Y$. Then $x^- \in \tau(C) \cap \tau(D)$.
   - Suppose $x \in D \cap C$ and $x \in X$. Then $x \in \tau(C) \cap \tau(D)$.
   - Suppose $\neg x \in D \cap C$ and $x \in X$. Then $\neg x \in \tau(C) \cap \tau(D)$.

   Hence $(\bigwedge \tau(D)) \models (\bigvee \tau(C))$ whence $(\bigwedge \tau(D)) \models \varphi'$ as required.
2. Let $C \in F$. Since $(\bigwedge D') \models \varphi'$ it follows that $(\bigwedge D') \models (\bigvee \tau(C))$.
   - Suppose $x^+ \in D' \cap \tau(C)$ and $x \in Y$. Then $x \in C \cap \tau^{-1}(D')$.
   - Suppose $x^- \in D' \cap \tau(C)$ and $x \in Y$. Then $\neg x \in C \cap \tau^{-1}(D')$.
   - Suppose $x \in D' \cap \tau(C)$ and $x \in X$. Then $x \in C \cap \tau^{-1}(D')$.
   - Suppose $\neg x \in D' \cap \tau(C)$ and $x \in X$. Then $\neg x \in C \cap \tau^{-1}(D')$.

   Hence $(\bigwedge \tau^{-1}(D')) \models (\bigvee C)$ whence $(\bigwedge \tau^{-1}(D')) \models \varphi$ as required.

□

The following corollary of the above relates implicants with literals drawn from Lit_Y to the satisfiability of the transformed clause set. It ultimately provides a correctness argument for the transformation map τ as used in the example.

Corollary 2.1 (Correctness)**.** *Let $F \subseteq \wp(\mathsf{Lit}_V)$ and let $\varphi = \bigwedge\{\bigvee C \mid C \in F\}$ denote a formula in CNF. Further, put $\varphi' = \bigwedge\{\bigvee \tau(C) \mid C \in F\}$. Then:*

- *If $D \in Cube_Y$ and $(\bigwedge D) \models \exists X : \varphi$ then $(\bigwedge \tau(D)) \wedge \varphi'$ is satisfiable.*
- *If $D' \in Cube_{\emptyset,Y}$ and $(\bigwedge D') \wedge \varphi'$ is satisfiable then $(\bigwedge \tau^{-1}(D')) \models \exists X : \varphi$.*

Let $\mathsf{sat}(\varphi) \subseteq \wp(V)$ denote the set of models of φ. The following result can the straightforwardly be adapted to the encoding $\tau_\ell(\varphi)$, which is identical to $\tau(\varphi)$ equipped with sorting networks. Observe that $\tau(\varphi)$ does not include any cardinality constraint on the size of the implicants, hence the need to define shortest implicants in terms of an implicant no longer than any other.

Corollary 2.2 (Shortest Implicants)**.** *Let $\varphi = \bigwedge\{\bigvee C \mid C \in F\}$ where $F \subseteq \wp(\mathsf{Lit}_V\}$. Then $D \in \mathsf{Cube}_Y$ is a shortest implicant of $\exists X : \varphi$ iff $D = \rho(M^\star \cap (Y^+ \cup Y^-))$ where*

1. *$M^\star \in \mathsf{sat}(\tau(\varphi))$*
2. *$|M^\star \cap (Y^+ \cup Y^-)| \leq |M \cap (Y^+ \cup Y^-)|$ for all $M \in \mathsf{sat}(\tau(\varphi))$*

2.2 Anytime Quantifier Elimination

This chapter shows how over-approximation using shortest prime implicants can be applied to eliminate $X = \{x_1, \ldots, x_n\}$ from $\exists X : \varphi$. In particular, we show how a SAT solver can be repeatedly called to compute a sequence of CNF formulae $h_0, h_1, \ldots$ that converge onto $\exists X : \varphi$ from above in the sense that $\exists X : \varphi$ entails each intermediate formula h_i. Furthermore, each h_{i+1} strictly entails h_i, so that the sequence is ultimately stationary. However, each h_i is free from all variables in X, hence this approach has the attractive property that generation of the sequence $h_0, h_1, \ldots, h_t$ can be stopped prematurely, at any time t, without sacrificing soundness. This leads to a so-called *anytime* (or *interruptible* [23, Sect. 2.6]) formulation of projection that compares, in some sense, favorably against resolution and model enumeration, which lead to all-or-nothing, monolithic approaches. Specifically, if $g_0 = \varphi$ and g_{i+1} is obtained from g_i by applying resolution to remove another variable $x_i \in X$ of from φ, then it is only the final formula g_n that is free from X. Moreover, the number of clauses in g_i does not necessarily decrease as i increases, and the size of intermediate g_i can be significantly larger than both φ and its projection g_n. By way of contrast, the size of the h_i increases monotonically as the sequence converges. We thus show how to systematically construct a sequence $h_0, h_1, \ldots, h_t$ which converges onto $\exists X : \varphi$ from above using shortest prime implicants.

2.2.1 Worked Example

The algorithm converges onto $\exists X : \varphi$ from above by incrementally conjoining clauses $\neg\nu_1, \ldots, \neg\nu_n$ formed from shortest prime implicants $\nu_1, \ldots, \nu_n$ of $\neg\exists X : \varphi$. Short clauses are likely to remove more models from the approximation than long ones, thereby encouraging rapid convergence. In this example, consider φ defined as:

$$\varphi \quad = \quad (\neg x \vee z) \wedge (y \vee z) \wedge (\neg x \vee \neg w \vee \neg z) \wedge (w \vee \neg z)$$

Put $X = \{z\}$ and $Y = \{w, x, y\}$. To derive an over-approximation of $\exists X : \varphi$, our algorithm operates on $\neg\varphi$ rather than φ directly. First, $\neg\varphi$ is turned into CNF using additional, fresh variables $T = \{t_1, \ldots, t_4\}$. Tseitin's method is used for this conversion, which yields a formula ψ. By construction, ψ is equisatisfiable to $\neg\varphi$:

$$\psi \quad = \quad \begin{cases} (x \vee t_1) & \wedge & (\neg z \vee t_1) & \wedge & & \\ (\neg y \vee t_2) & \wedge & (\neg z \vee t_2) & \wedge & & \\ (x \vee t_3) & \wedge & (w \vee t_3) & \wedge & (z \vee t_3) & \wedge \\ (\neg w \vee t_4) & \wedge & (z \vee t_4) & \wedge & (\neg t_1 \vee \neg t_2 \vee \neg t_3 \vee \neg t_4) & \end{cases}$$

The Tseitin variable t_i indicates whether a truth assignment violates the i^{th} cube of $\neg\varphi$. Applying the transformation τ given in Def. 2.1 then yields:

$$\tau(\psi) \quad = \quad \begin{cases} (x^+ \vee t_1) & \wedge & (\neg z \vee t_1) & \wedge & & \\ (y^- \vee t_2) & \wedge & (\neg z \vee t_2) & \wedge & & \\ (x^+ \vee t_3) & \wedge & (w^+ \vee t_3) & \wedge & (z \vee t_3) & \wedge \\ (w^- \vee t_4) & \wedge & (z \vee t_4) & \wedge & (\neg t_1 \vee \neg t_2 \vee \neg t_3 \vee \neg t_4) & \wedge \\ (\neg w^+ \vee \neg w^-) & \wedge & (\neg x^+ \vee \neg x^-) & \wedge & (\neg y^+ \vee \neg y^-) & \end{cases}$$

Existential Quantification by Filtering

To see how $\tau(\psi)$ can be applied to find an over-approximation of $\exists X : \varphi$, observe:

$$\begin{aligned} \nu \models \forall X : \exists T : \psi \quad &\Leftrightarrow \quad \neg\forall X : \exists T : \psi \models \neg\nu \\ &\Leftrightarrow \quad \exists X : \neg\exists T : \psi \models \neg\nu \\ &\Leftrightarrow \quad \exists X : \varphi \models \neg\nu \end{aligned}$$

Hence, to find an over-approximation of $\exists X : \varphi$, it suffices to find an implicant of $\forall X : \exists T : \psi$. To find such an implicant, observe that $\forall X : \exists T : \psi \models \exists X : \exists T : \psi$. Every implicant ν of $\forall X : \exists T : \psi$ is thus also an implicant of $\exists X : \exists T : \psi$. Since the τ map allows us to derive implicants of $\exists X : \exists T : \psi$, this suggests a strategy in which the implicants of $\exists X : \exists T : \psi$ are filtered to find the implicants of $\forall X : \exists T : \psi$. The check $\exists X : \varphi \models \neg\nu$ amounts to deciding whether the conjoined formula $\varphi \wedge \nu$ is

unsatisfiable. Thus, a test for unsatisfiability can be used for filtering. To illustrate, suppose that a SAT solver produces the following solution to the formula $\tau(\psi)$:

$$\mathbf{m}_1 = \left\{ \begin{array}{cccccccccccc} w^+ & \mapsto & 0, & w^- & \mapsto & 1, & x^+ & \mapsto & 0, & x^- & \mapsto & 0, \\ y^+ & \mapsto & 0, & y^- & \mapsto & 0, & z & \mapsto & 1, & & & \\ t_1 & \mapsto & 1, & t_2 & \mapsto & 1, & t_3 & \mapsto & 1, & t_4 & \mapsto & 0 \end{array} \right\}$$

The cube $\rho_Y(\mathbf{m}_1) = (\neg w)$ entails $\exists X : \exists T : \psi$, and therefore, it remains to check whether $\exists X : \varphi \models \neg\rho_Y(\mathbf{m}_1)$. Since $\varphi \wedge \rho_Y(\mathbf{m}_1)$ is satisfiable, the cube is discarded; it does not entail $\forall X : \exists T : \psi$. However, before discarding the cube, the formula $\tau(\psi)$ is augmented with $(\neg w^- \vee x^+ \vee x^- \vee y^- \vee y^+)$ to prevent $\rho_Y(\mathbf{m}_1)$ from being found again. This blocking clause can be interpreted as an implication $w^- \rightarrow (x^+ \vee x^- \vee y^- \vee y^+)$, which ensures that any cube subsequently found that entails $\rho_Y(\mathbf{m}_1)$ also has more literals than $\rho_Y(\mathbf{m}_1)$. Applying a SAT solver then yields a model

$$\mathbf{m}_2 = \left\{ \begin{array}{cccccccccccc} w^+ & \mapsto & 0, & w^- & \mapsto & 0, & x^+ & \mapsto & 1, & x^- & \mapsto & 0, \\ y^+ & \mapsto & 0, & y^- & \mapsto & 0, & z & \mapsto & 0, & & & \\ t_1 & \mapsto & 0, & t_2 & \mapsto & 1, & t_3 & \mapsto & 1, & t_4 & \mapsto & 0 \end{array} \right\}$$

and hence $\rho_Y(\mathbf{m}_2) = (x)$. Since $\varphi \wedge \rho_Y(\mathbf{m}_2)$ is unsatisfiable, we conclude that $\exists X : \varphi \models \neg\rho_Y(\mathbf{m}_2)$, hence $\neg\rho_Y(\mathbf{m}_2)$ constitutes an over-approximation of $\exists X : \varphi$. The blocking clause $\neg x^+$ is then added to $\tau(\psi)$ to prevent any cube that entails $\rho_Y(\mathbf{m}_2)$ being found. Note that this blocking clause differs in structure from the one imposed previously. Indeed, the number of literals in the clause is merely n where n is the number of literals in the cube. In the previous case, the number of literals in the blocking clause is $2 \cdot |Y| - n$. Reapplying a SAT solver yields a further model

$$\mathbf{m}_3 = \left\{ \begin{array}{cccccccccccc} w^+ & \mapsto & 0, & w^- & \mapsto & 1, & x^+ & \mapsto & 0, & x^- & \mapsto & 0, \\ y^+ & \mapsto & 0, & y^- & \mapsto & 1, & z & \mapsto & 1, & & & \\ t_1 & \mapsto & 1, & t_2 & \mapsto & 1, & t_3 & \mapsto & 1, & t_4 & \mapsto & 0 \end{array} \right\}$$

which defines the cube $\rho_Y(\mathbf{m}_3) = (\neg w \wedge \neg y)$. Since $\varphi \wedge \rho_Y(\mathbf{m}_3)$ is unsatisfiable, it again follows that $\exists X : \varphi \models \neg\rho_Y(\mathbf{m}_3)$, which refines the over-approximation of $\exists X : \varphi$ to the conjunction $(\neg\rho_Y(\mathbf{m}_2)) \wedge (\neg\rho_Y(\mathbf{m}_3))$. The blocking clause $(\neg w^- \vee \neg y^-)$ is then added to the augmented formula at which point one final application of the solver indicates that the conjoined formula is unsatisfiable. Hence, convergence onto $\exists X : \varphi$ has been obtained from above where:

$$\begin{array}{rcl} \exists X : \varphi & = & \neg\rho_Y(\mathbf{m}_2) \wedge \neg\rho_Y(\mathbf{m}_3) \\ & = & (\neg x) \wedge (w \vee y) \end{array}$$

Terminating the procedure early, before $\rho_Y(\mathbf{m}_3)$ is computed, would yield the over-approximation $\neg\rho_Y(\mathbf{m}_2) = (\neg x)$ which, though safe, has strictly more models

than $(\neg x) \wedge (w \vee y)$. Thus, the method is diametrically opposed to resolution: In the resolution-based scheme, the projection is found in the last step only after all variables have been eliminated one after the other. In the above SAT-based scheme, a clause in the projection space is obtained in the first step, as in a parallel form of elimination, which is subsequently refined by adding further clauses.

Search-Space Reduction using Instantiation

In the example discussed so far, a SAT solver generates a spurious candidate $\rho_Y(\mathbf{m}_1)$ for an implicant, which is then refuted by checking $\varphi \models \neg\rho_Y(\mathbf{m}_1)$. This scheme is based on the observation that every implicant of $\forall X : \exists T : \psi$ is also an implicant of $\exists X : \exists T : \psi$, where in the case of the example $X = \{z\}$. However, observe that

$$\forall X : \exists T : \psi \quad \models \quad \exists T : \psi[z \mapsto 0]$$

where $\psi[z \mapsto 0]$ denotes the formula obtained by replacing each occurrence of z in ψ with the truth value 0. We refer to this operation as instantiation. Therefore, every implicant of $\forall X : \exists T : \psi$ is also an implicant of $\exists T : \psi[z \mapsto 0]$. The formula $\exists T : \psi[z \mapsto 0]$ is not only a simplification of $\exists T : \psi$, but $\exists T : \psi[z \mapsto 0]$ will possess fewer models, and hence fewer implicants, than $\exists X : \exists T : \psi$ provided $\psi \not\models \neg z$. Consider again the formula $\tau(\psi)$ with the instance $\tau(\psi[z \mapsto 0]) = \tau(\psi)[z \mapsto 0]$. Recall that originally the candidate implicant $\rho_Y(\mathbf{m}_1) = (\neg w)$ was derived, which was then refuted because $\exists X : \varphi \not\models \neg\rho_Y(\mathbf{m}_1)$. This candidate is suppressed by the instantiation and is not a model of $\tau(\psi)[z \mapsto 0]$. It turns out that 13 SAT instances are required to converge onto $\exists X : \varphi$, whereas operating on $\tau(\psi)[z \mapsto 0]$ and $\tau(\psi)[z \mapsto 1]$ only requires 9 and 10 SAT instances, respectively. Interestingly, the formulae derived for these cases are equivalent but different. For $\tau(\psi)[z \mapsto 0]$, we obtain the limit $(\neg x) \wedge (w \vee y)$ as expected, but operating on $\tau(\psi)[z \mapsto 1]$ yields $(w \vee y) \wedge (w \vee \neg x) \wedge (\neg w \vee \neg x)$. This result is equivalent to $(\neg x) \wedge (w \vee y)$ as $(w \vee \neg x) \wedge (\neg w \vee \neg x)$ can be simplified to $(\neg x)$.

Search-Space Reduction using Multiple Instantiations

Instantiating the variables of X with truth values can decrease the number of spurious implicants that are generated. This suggests instantiating ψ in different ways, then combining the instantiations to limit the search space. Thus, the basic idea is to derive multiple instantiations, say, $\tau(\psi)[z \mapsto 0]$ and $\tau(\psi)[z \mapsto 1]$ and solve:

$$\mu \quad = \quad \tau(\psi)[z \mapsto 0] \wedge \tau(\psi)[z \mapsto 1]$$

In case of $|X| = 1$, μ is equivalent to $\forall X : \tau(\psi)$. Care is needed to avoid accidental coupling between the Tseitin variables in different instantiations. This can be

avoided by introducing fresh, disjoint sets of variables $T_1 = \{t_{i,1} \mid t_i \in T\}$ and $T_2 = \{t_{i,2} \mid t_i \in T\}$ and by applying substitutions to $\tau(\psi)[z \mapsto 0]$ and $\tau(\psi)[z \mapsto 1]$, respectively. By applying simplification, we obtain:

$$\mu \quad = \quad \begin{cases} (x^+ \vee t_{1,1}) & \wedge & (y^- \vee t_{1,2}) & \wedge & (\neg t_{1,1} \vee \neg t_{1,2}) & \wedge \\ (x^+ \vee t_{2,3}) & \wedge & (w^+ \vee t_{2,3}) & \wedge & (w^- \vee t_{2,4}) & \wedge \\ (\neg t_{2,3} \vee \neg t_{2,4}) & \wedge & & & & \\ (\neg w^+ \vee \neg w^-) & \wedge & (\neg x^+ \vee \neg x^-) & \wedge & (\neg y^+ \vee \neg y^-) & \end{cases}$$

When solving for μ, the sequence of over-approximations converges onto the limit $(w \vee y) \wedge (w \vee \neg x) \wedge (\neg w \vee \neg x)$ without encountering any spurious implicants. Of course, since $|X| = 1$, this is because $\forall X : \exists T : \psi \equiv (\exists T : \psi[z \mapsto 0]) \wedge (\exists T : \psi[z \mapsto 1])$. Interestingly, μ consists of 10 clauses whereas $\tau(\psi)$ has 13 clauses. This is because instantiating X confers significant opportunities for simplification, offering scope for applying multiple instantiation without generating a formula that is unwieldy.

2.2.2 Formal Correctness

The technique presented in the example rests on finding implicants of $\neg\exists X : \varphi$ by operating over $\exists X : \exists T : \psi$ and keeping those that imply $\forall X : \exists T : \psi$. The transformation τ reduces this problem to SAT. Although some correctness arguments have been weaved into the example, this section studies correctness with more rigor. First of all, we show that the strategy of generating implicants of $\exists X : \neg\varphi$ and eliminating those which are not implied by $\forall X : \neg\varphi$ is indeed correct.

Proposition 2.2 (Correctness)**.** *Let $\varphi = \bigwedge\{\bigvee C \mid C \in F\}$ where $F \subseteq \wp(\mathsf{Lit}_V)$, and let $D \in \mathsf{Cube}_Y$. Then $\exists X : \varphi \models \neg D$ iff the following two conditions hold:*

1. *$D \models \exists X : \neg\varphi$*
2. *φ and D are inconsistent.*

Proof. We prove the equivalence as two implications.

1. Suppose $\exists X : \varphi \models \neg D$, hence $D \models \neg\exists X : \varphi$, which is equivalent to $D \models \forall X : \neg\varphi$. Since $\mathsf{sat}(\forall X : \neg\varphi) \subseteq \mathsf{sat}(\exists X : \varphi)$, we have $D \models \exists X : \neg\varphi$. It remains to show that φ and D are inconsistent, which is equivalent to $\varphi \wedge D$ being unsatisfiable. This follows from $\mathsf{sat}(\exists X : \varphi) \subseteq \mathsf{sat}(\neg D) = \wp(\mathsf{Lit}_V) \setminus \mathsf{sat}(D)$.
2. Suppose $D \models \exists X : \neg\varphi$ and $\varphi \wedge D$ is unsatisfiable, hence $\mathsf{sat}(\varphi) \cap \mathsf{sat}(D) = \emptyset$ and thus $\mathsf{sat}(\exists X : \varphi) \cap \mathsf{sat}(D) = \emptyset$. Then $\neg\exists X : \neg\varphi \models \neg D$, hence $\forall X : \neg\varphi \models \neg D$. Assume $\exists X : \varphi \not\models \neg D$, then $\mathsf{sat}(\exists X : \varphi) \not\subseteq \wp(\mathsf{Lit}_V) \setminus \mathsf{sat}(D)$, which implies $\mathsf{sat}(\exists X : \varphi) \cap \mathsf{sat}(D) \neq \emptyset$, a contradiction.

□

Correctness of the construction follows directly from Prop. 2.2 and Cor. 2.2. We further show that instantiation does not affect correctness.

Proposition 2.3 (Instantiation)**.** *Let* $\varphi = \bigwedge\{\bigvee C \mid C \in F\}$ *where* $F \subseteq \wp(\mathsf{Lit}_V)$*. Let* $D \in \mathsf{Cube}_Y$ *such that* $\exists X : \varphi \models \neg D$*,* $x \in X$*, and* $c \in \{0, 1\}$*. Then* $\forall X : \varphi \models \exists X : (\varphi[x \mapsto c]) \models \exists X : \varphi$*.*

Proof. Since $\mathsf{sat}(\varphi[x \mapsto c]) = \mathsf{sat}(\varphi) \cap \wp(\mathsf{Lit}_V \mid x = c)$, we have $\mathsf{sat}(\varphi[x \mapsto c]) \subseteq \mathsf{sat}(\varphi)$, which is equivalent to $\varphi[x \mapsto c] \models \varphi$. It follows that $\exists X : \varphi[x \mapsto c] \models \exists X : \varphi$ from monotonicity of projection. It holds true that $\forall X : \varphi = \forall X \setminus \{x\} : \forall x : \varphi$ is equivalent to $(\forall X \setminus \{x\} : \varphi[x \mapsto 0]) \wedge (\forall X \setminus \{x\} : \varphi[x \mapsto 1])$, hence $\mathsf{sat}(\forall X : \varphi) = \mathsf{sat}(\forall X \setminus \{x\} : \varphi[x \mapsto 0]) \cap \mathsf{sat}(\forall X \setminus \{x\} : \varphi[x \mapsto 1])$. It follows that $\mathsf{sat}(\forall X : \varphi) \subseteq \mathsf{sat}(\forall X \setminus \{x\} : \varphi[x \mapsto c])$ for $c \in \{0, 1\}$. □

An immediate consequence of Prop. 2.3 is that multiple instantiation does not affect correctness either.

2.3 Two-Phase Quantifier Elimination

The algorithm for quantifier elimination discussed in Chap. 2.2 features the interesting property that it is interruptible. This suggests that it could be applied in case computing the quantifier-free formula is intractable. Approximations may still deliver useful results. However, this property comes at a price. Candidate implicants are computed — in theory, there can be exponentially many of which — and a satisfiability check is then required to filter spurious candidates. The two-phase algorithm presented in this chapter contrasts with the anytime algorithm in that it possesses the "everyone a winner" [183] enumeration property. This means that, rather than enumerating and filtering potential clauses of $\exists X : \varphi$, a new clause of $\exists X : \varphi$ is found on (virtually) each application of a SAT solver. This property is highly desirable, because it couples the computational effort required to compute the quantifier-free version of φ with its size.

2.3.1 Worked Example

As before, let φ denote a quantifier-free Boolean formula that ranges over sets X and Y of propositional variables. The key idea behind the two-phase method is to first converge onto the set of solutions of the formula $\exists X : \varphi$ from below using implicants. The first phase gives a DNF formula $\bigvee_{i=1}^{n} c_i$ equivalent to $\exists X : \varphi$, followed by a second phase that converges onto a CNF representation of $\exists X : \varphi$ from above, based on $\bigvee_{i=1}^{n} c_i$. We build towards this technique using $\varphi = \xi \wedge \mu$ from the introduction.

Enumerating Implicants

The first step of our method is to enumerate the implicants of φ in the projection space. To do so, we first convert φ into CNF, for which we introduce a set of Tseitin variables T as before. The T variables are existentially quantified, and the resulting formula ψ in CNF is equisatisfiable to φ. Introducing fresh variables ensures that the size of ψ is only a linear multiple of the size of φ. As before $\tau(\psi)$ refers to the syntactic transformation defined over the variables $V = X \cup Y^+ \cup Y^- \cup T$. Passing $\tau(\psi)$ to a SAT solver yields a model $\mathbf{m}_1 : V \to \mathbb{B}$, such as:

$$\mathbf{m}_1 \;=\; \left\{ \begin{array}{llllll} x_1 \mapsto 1, & x_2 \mapsto 0, & x_3 \mapsto 1, & x_4 \mapsto 0, & x_5 \mapsto 1, & x_6 \mapsto 0 \\ y_1^+ \mapsto 0, & y_2^+ \mapsto 0, & y_3^+ \mapsto 0, & y_4^+ \mapsto 1, & y_5^+ \mapsto 0, & y_6^+ \mapsto 1 \\ y_1^- \mapsto 1, & y_2^- \mapsto 1, & y_3^- \mapsto 1, & y_4^- \mapsto 0, & y_5^- \mapsto 1, & y_6^- \mapsto 0 \end{array} \right\}$$

The variables in $\mathbf{m}_1 \cap (Y^+ \cup Y^-)$ then define a conjunction of literals, a cube, $\rho(\mathbf{m}_1)$ over the variables in Y, which is given as:

$$\begin{array}{rcl} \rho(\mathbf{m}_1) & = & \big(\bigwedge\{y_i \mid y_i^+ \in (\mathbf{m}_1 \cap Y^+)\}\big) \wedge \big(\bigwedge\{\neg y_i \mid y_i^- \in (\mathbf{m}_1 \cap Y^-)\}\big) \\ & = & (\neg y_1 \wedge \neg y_2 \wedge \neg y_3 \wedge y_4 \wedge \neg y_5 \wedge y_6) \end{array}$$

The cube $\rho(\mathbf{m}_1)$ is an implicant of $\exists X : \varphi$ since $\rho(\mathbf{m}_1) \models \exists X : \varphi$. It constitutes an under-approximation of $\exists X : \varphi$ since the set of all models of $\rho(\mathbf{m}_1)$ is a subset of the set of all models of $\exists X : \varphi$. To find another under-approximation, and specifically one that is not itself entailed by $\rho(\mathbf{m}_1)$, we augment $\tau(\psi)$ with the blocking clause:

$$\begin{array}{rcl} \beta(\mathbf{m}_1) & = & \big(\bigvee\{y_i^- \mid y_i^+ \in (\mathbf{m}_1 \cap Y^+)\}\big) \vee \big(\bigvee\{y_i^+ \mid y_i^- \in (\mathbf{m}_1 \cap Y^-)\}\big) \\ & = & (y_1^+ \vee y_2^+ \vee y_3^+ \vee y_4^- \vee y_5^+ \vee y_6^-) \end{array}$$

Of course, enumerating implicants in this way dovetails with the advances in incremental SAT. Applying a solver to the augmented formula $\tau(\psi)' = \tau(\psi) \wedge \beta(\mathbf{m}_1)$ gives another model $\mathbf{m}_2$ as follows:

$$\mathbf{m}_2 \;=\; \left\{ \begin{array}{llllll} x_1 \mapsto 1, & x_2 \mapsto 0, & x_3 \mapsto 1, & x_4 \mapsto 0, & x_5 \mapsto 1, & x_6 \mapsto 0 \\ y_1^+ \mapsto 0, & y_2^+ \mapsto 1, & y_3^+ \mapsto 0, & y_4^+ \mapsto 1, & y_5^+ \mapsto 0, & y_6^+ \mapsto 1 \\ y_1^- \mapsto 1, & y_2^- \mapsto 0, & y_3^- \mapsto 1, & y_4^- \mapsto 0, & y_5^- \mapsto 1, & y_6^- \mapsto 0 \end{array} \right\}$$

The model $\mathbf{m}_2$ defines another implicant $\rho(\mathbf{m}_2) = (\neg y_1 \wedge y_2 \wedge \neg y_3 \wedge y_4 \wedge \neg y_5 \wedge y_6)$ of $\exists X : \varphi$, hence $\rho(\mathbf{m}_1) \vee \rho(\mathbf{m}_2) \models \exists X : \varphi$. Repeating this strategy to derive implicants yields an unsatisfiable formula after the fourth step, and thus

$$\bigvee_{i=1}^{4} \rho(\mathbf{m}_i) \;=\; \left\{ \begin{array}{llllllllll} (\neg y_1 & \wedge & \neg y_2 & \wedge & \neg y_3 & \wedge & y_4 & \wedge & \neg y_5 & \wedge & y_6) & \vee \\ (\neg y_1 & \wedge & y_2 & \wedge & \neg y_3 & \wedge & y_4 & \wedge & \neg y_5 & \wedge & y_6) & \vee \\ (y_1 & \wedge & \neg y_2 & \wedge & \neg y_3 & \wedge & y_4 & \wedge & \neg y_5 & \wedge & y_6) & \vee \\ (y_1 & \wedge & \neg y_2 & \wedge & y_3 & \wedge & \neg y_4 & \wedge & y_5 & \wedge & \neg y_6) & \end{array} \right.$$

satisfies $\bigvee_{i=1}^{4} \rho(\mathbf{m}_i) = \exists X : \varphi$. However, observe that $\bigvee_{i=1}^{4} \rho(\mathbf{m}_i)$ is in DNF and also contains redundancies, such as:

$$\rho(\mathbf{m}_1) \vee \rho(\mathbf{m}_2) \quad = \quad (\neg y_1 \wedge \neg y_3 \wedge y_4 \wedge \neg y_5 \wedge y_6)$$

Cardinality constraints using sorting networks can — as we have already demonstrated in Chap. 2.1.2 — be used to eliminate such redundancies. Computing implicants using $\tau_\ell(\psi)$ rather than $\tau(\psi)$ yields a smaller DNF formula, based on three models $\mathbf{m}'_1$, $\mathbf{m}'_2$, and $\mathbf{m}'_3$:

$$\bigvee_{i=1}^{3} \rho(\mathbf{m}'_i) \quad = \quad \left\{ \begin{array}{lllllllllll} (\neg y_1 & \wedge & \neg y_3 & \wedge & y_4 & \wedge & \neg y_5 & \wedge & y_6) & & \vee \\ (y_1 & \wedge & \neg y_2 & \wedge & \neg y_3 & \wedge & y_4 & \wedge & \neg y_5 & \wedge & y_6) \quad \vee \\ (y_1 & \wedge & \neg y_2 & \wedge & y_3 & \wedge & \neg y_4 & \wedge & y_5 & \wedge & \neg y_6) \end{array} \right.$$

Over-Approximation by Dualization

Recall that we are interested in obtaining CNF, whereas the construction we have presented so far yields formulae in DNF. Direct conversion of a formula in DNF to an equivalent one in CNF may increase the size of the formula exponentially. However, observe that since $\exists X : \varphi = \bigvee_{i=1}^{3} \rho(\mathbf{m}'_i)$, we have:

$$\neg\exists X : \varphi \quad = \quad \neg \bigvee_{i=1}^{3} \rho(\mathbf{m}'_i) \quad = \quad \bigwedge_{i=1}^{3} \neg\rho(\mathbf{m}'_i)$$

Latter formula can be converted into CNF straightforwardly by pushing negations inward. We can thus reapply the above construction to infer implicants of $\neg\exists X : \varphi$. Given a cube ν such that $\nu \models \neg\exists X : \varphi$, the contrapositive holds, giving $\exists X : \varphi \models \neg\nu$. Therefore $\neg\nu$ over-approximates $\exists X : \varphi$. In order to apply the above method on the dual of $\bigvee_{i=1}^{3} \rho(\mathbf{m}'_i)$, we start by negating the formula to give:

$$\neg\exists X : \varphi \quad = \quad \left\{ \begin{array}{ll} (y_1 \vee y_3 \vee \neg y_4 \vee y_5 \vee \neg y_6) & \wedge \\ (\neg y_1 \vee y_2 \vee y_3 \vee \neg y_4 \vee y_5 \vee \neg y_6) & \wedge \\ (\neg y_1 \vee y_2 \vee \neg y_3 \vee y_4 \vee \neg y_5 \vee y_6) & \end{array} \right.$$

Denote this formula by ω and apply τ to ω to give:

$$\tau(\omega) \quad = \quad \left\{ \begin{array}{lll} (y_1^- \vee y_3^- \vee y_4^+ \vee y_5^+ \vee y_6^+) & \wedge & \\ (y_1^+ \vee y_2^- \vee y_3^- \vee y_4^+ \vee y_5^- \vee y_6^+) & \wedge & \\ (y_1^+ \vee y_2^- \vee y_3^+ \vee y_4^- \vee y_5^+ \vee y_6^-) & \wedge & (\bigwedge_{i=1}^{6} \neg(y_i^+ \wedge y_i^-)) \end{array} \right.$$

We then solve $\tau_1(\omega)$, which is unsatisfiable: $\neg \bigvee_{i=1}^{3} \rho(\mathbf{m}'_i)$ does not posses implicants of length 1. Passing $\tau_2(\omega)$ to a SAT solver yields a model $\mathbf{m}''_1$ as follows:

$$\mathbf{m}''_1 \quad = \quad \left\{ \begin{array}{llllll} y_1^+ \mapsto 0, & y_2^+ \mapsto 1, & y_3^+ \mapsto 0, & y_4^+ \mapsto 0, & y_5^+ \mapsto 0, & y_6^+ \mapsto 0 \\ y_1^- \mapsto 0, & y_2^- \mapsto 0, & y_3^- \mapsto 0, & y_4^- \mapsto 0, & y_5^- \mapsto 0, & y_6^- \mapsto 1 \end{array} \right\}$$

We then extract a cube $\rho(\mathbf{m}_1'') = (y_2 \wedge \neg y_6)$ from $\mathbf{m}_1''$. From $\rho(\mathbf{m}_1'') \models \neg\exists X : \varphi$, we deduce $\exists X : \varphi \models \neg\rho(\mathbf{m}_1'')$. Since $\rho(\mathbf{m}_1'')$ is a cube, $\neg\rho(\mathbf{m}_1'')$ clearly is a clause. Thus, $\neg\rho(\mathbf{m}_1'')$ can directly be added to the SAT instance as a blocking clause, denoted $\beta(\mathbf{m}_1'')$. We add this blocking clause to suppress the cube as before and retrieve a model $\mathbf{m}_2''$ for $\tau_2(\omega) \wedge \beta(\mathbf{m}_1'')$ from the SAT solver:

$$\mathbf{m}_2'' \;=\; \left\{ \begin{array}{lllllll} y_1^+ \mapsto 0, & y_2^+ \mapsto 0, & y_3^+ \mapsto 1, & y_4^+ \mapsto 0, & y_5^+ \mapsto 0, & y_6^+ \mapsto 1 \\ y_1^- \mapsto 0, & y_2^- \mapsto 0, & y_3^- \mapsto 0, & y_4^- \mapsto 0, & y_5^- \mapsto 0, & y_6^- \mapsto 0 \end{array} \right\}$$

This model induces a cube $\rho(\mathbf{m}_2'') = (y_3 \wedge y_6)$. Then, $\exists X : \varphi \models \neg\rho(\mathbf{m}_1'') \wedge \neg\rho(\mathbf{m}_2'')$ and $\tau_2(\omega) \wedge \beta(\mathbf{m}_1'') \wedge \beta(\mathbf{m}_2'')$ is unsatisfiable. We proceed with cubes of length 3 and solve $\tau_3(\omega) \wedge \beta(\mathbf{m}_1'') \wedge \beta(\mathbf{m}_2'')$, which gives rise to a cube $\rho(\mathbf{m}_3'') = (\neg y_2 \wedge \neg y_5 \wedge \neg y_6)$. By adding blocking clauses and enumerating all cubes $\rho(\mathbf{m}_i'')$ for $i \in \{1, \ldots, m\}$, we could derive a CNF formula $\bigwedge_{i=1}^{m} \neg\rho(\mathbf{m}_i'')$ equivalent to $\exists X : \varphi$.

Relaxed Cubes

However, we can improve on this naïve strategy and produce a denser CNF representation by searching for a shorter sub-cube c' of $\rho(\mathbf{m}_3'')$ which is itself an implicant of ω. The cube c' that we aim to compute is weaker in the sense that it satisfies:

- $\mathsf{vars}(c') \subset \mathsf{vars}(\rho(\mathbf{m}_3''))$
- $\mathsf{sat}(\rho(\mathbf{m}_3'')) \subset \mathsf{sat}(c')$

Yet, c' obeys the requirement $c' \models \omega$. We refer to the computation of such a cube c' as *relaxation* or *weakening*. To do this, let:

$$\begin{array}{rcl} N & = & (Y^+ \cup Y^-) \setminus \mathbf{m}_3'' \\ & = & \{y_1^+, y_1^-, y_2^+, y_3^+, y_3^-, y_4^-, y_5^+, y_6^+\} \end{array}$$

We then solve $\tau_2(\omega)$ in conjunction with the cube

$$\bigwedge\{\neg y_i^+ \mid y_i^+ \in N \cap Y^+\} \wedge \bigwedge\{\neg y_i^- \mid y_i^- \in N \cap Y^-\}$$

which we pass the solver as an assumption. The solver produces a model

$$\mathbf{m}_4'' \;=\; \left\{ \begin{array}{lllllll} y_1^+ \mapsto 0, & y_2^+ \mapsto 0, & y_3^+ \mapsto 0, & y_4^+ \mapsto 0, & y_5^+ \mapsto 0, & y_6^+ \mapsto 0 \\ y_1^- \mapsto 0, & y_2^- \mapsto 0, & y_3^- \mapsto 0, & y_4^- \mapsto 0, & y_5^- \mapsto 1, & y_6^- \mapsto 1 \end{array} \right\}$$

which defines $\rho(\mathbf{m}_4'') = (\neg y_5 \wedge \neg y_6)$; thus $\neg\rho(\mathbf{m}_4'') = (y_5 \vee y_6) \models \exists X : \varphi$. Since

$$\mathbf{m}_4'' \cap (Y^+ \cup Y^-) \;\subset\; \mathbf{m}_3'' \cap (Y^+ \cup Y^-)$$

we have $\mathbf{m}_3'' \models \mathbf{m}_4''$ and $\rho(\mathbf{m}_3'') \models \rho(\mathbf{m}_4'')$. We thus discard $\rho(\mathbf{m}_3'')$ and proceed with:

$$\tau_3(\omega) \wedge \beta(\mathbf{m}_1'') \wedge \beta(\mathbf{m}_2'') \wedge \beta(\mathbf{m}_4'')$$

Whenever a fresh cube is discovered, we apply the same strategy to relax it to the most general one that still entails ω. It is interesting to note that an implicant of length ℓ can be generalized using at most $\lceil \log_2(\ell) \rceil$ calls to a solver by applying dichotomic search, although we do not apply this optimization since ℓ is typically small. Repeatedly applying this generalization scheme, we derive the following minimal (though not unique) CNF representation of $\exists X : \varphi$ in five more iterations:

$$\exists X : \varphi \quad = \quad \left\{ \begin{array}{lllllllll} (\neg y_2 \vee y_6) & \wedge & (\neg y_3 \vee \neg y_6) & \wedge & (y_5 \vee y_6) & \wedge & (y_3 \vee \neg y_5) & \wedge \\ (y_4 \vee \neg y_6) & \wedge & (y_1 \vee y_6) & \wedge & (\neg y_1 \vee \neg y_2) & \wedge & (\neg y_4 \vee y_6) & \end{array} \right.$$

Since the search is exhaustive, this is no longer an over-approximation of the projection, but equivalent to it. Our implementation using MiniSat takes 0.0012s and 0.0009s for the first and second stages of the algorithm, taking 0.0021s overall.

2.3.2 Formal Correctness

To state how to compute an image by enumerating implicants, we formalize the unusual notion of a blocking clause.

Definition 2.4 (Blocking Clause)**.** *The map* $\beta : \mathsf{Cube}_{X,Y} \to \mathsf{Cube}_{\emptyset,Y}$ *is defined as:*

$$\beta(D') \quad = \quad \{y_i^- \mid y_i^- \in D'\} \cup \{y_i^+ \mid y_i^+ \in D'\}$$

Theorem 2.1 (Correctness)**.** *Let* $\varphi = \bigwedge\{\bigvee C \mid C \in F\}$ *where* $F \subseteq \wp(\mathsf{Lit}^V)$ *and put* $\varphi' = \bigwedge\{\bigvee \tau(C) \mid C \in F\}$*. Let* $D_1', \ldots, D_n' \in \mathsf{Cube}_{\emptyset,Y}$ *be a sequence such that*

- $(\bigwedge D_k') \wedge \varphi' \wedge \bigwedge_{i=1}^{k-1}(\bigvee \beta(D_i'))$ *is satisfiable for all* $1 \leq k \leq n$*, and*
- $\varphi' \wedge \bigwedge_{i=1}^{n}(\bigvee \beta(D_i'))$ *is unsatisfiable.*

Then, $\bigvee_{i=1}^{n} \tau^{-1}(D_i') = \exists X : \varphi$*.*

Proof. We prove both statements separately.

- Let $k \in \{1, \ldots, l\}$. Since $\bigwedge D_k' \wedge \varphi' \wedge (\bigwedge_{i=1}^{k-1} \beta(D_i'))$ is satisfiable and $\bigwedge D_k' \wedge \varphi' \wedge (\bigwedge_{i=1}^{k-1} \beta(D_i')) \models \bigwedge D_k' \wedge \varphi'$, it follows that $\bigwedge D_k' \wedge \varphi'$ is satisfiable. Hence, by Cor. 2.1, $\bigwedge \tau^{-1}(D_k') \models \exists X : \varphi$ whence $\bigvee_{i=1}^{l} \bigwedge \tau^{-1}(D_i') \models \exists X : \varphi$.

- Suppose there exists $D \in \mathsf{Cube}_Y$ such that $D \models \exists X : \varphi \wedge \neg(\bigvee_{i=1}^{l} \bigwedge \tau^{-1}(D'_i))$. Then:

$$\begin{array}{rcl} D & \models & \neg(\bigvee_{i=1}^{l} \bigwedge \tau^{-1}(D'_i)) \\ & = & \bigwedge_{i=1}^{l} \neg \bigwedge \tau^{-1}(D'_i) \\ & = & \bigwedge_{i=1}^{l} \bigvee \tau^{-1}(\beta(D'_i)) \end{array}$$

  Thus, we have $\tau(D) \models \bigwedge_{i=1}^{l}(\bigvee \beta(D'_i))$ and $\tau(D) \wedge \varphi'$ is satisfiable. Therefore, $\tau(D) \wedge \varphi' \wedge \bigwedge_{i=1}^{l}(\bigvee \beta(D'_i))$ is satisfiable, a contradiction, and thus $\exists X : \varphi \models \bigvee_{i=1}^{l}(\bigwedge \tau^{-1}(D'_i))$.

□

The following proposition squares with the correctness result to show how a CNF representation of the projection can be derived in a two-phase process.

Proposition 2.4 (Dualization)**.** *Let* $\psi = \bigvee_{i=1}^{n} (\bigwedge D_i)$ *where* $D_1, \ldots, D_n \in \mathsf{Cube}_Y$*. Further, let* $\exists X : \varphi = \bigvee_{i=1}^{m} (\bigwedge E_i)$ *where* $E_1, \ldots, E_m \in \mathsf{Cube}Y$ *and define* $\varphi = \bigwedge_{i=1}^{n} (\bigvee_{l \in D_i} \neg l)$*. Then,* $\psi = \bigwedge_{i=1}^{m} (\bigvee_{l \in E_i} \neg l)$*.*

The corollary that follows is an immediate consequence and states that the second phase can be aborted prematurely without sacrificing correctness.

Corollary 2.3 (Anytime)**.** *Let* $\psi = \bigvee_{i=1}^{n} (\bigwedge D_i)$ *where* $D_1, \ldots, D_n \in \mathsf{Cube}_Y$*. Let* $\bigwedge_{i=1}^{m} E_i \models \exists X : \varphi$ *where* $E_1, \ldots, E_m \in \mathsf{Cube}_Y$ *and* $\varphi = \bigwedge_{i=1}^{n} (\bigvee_{l \in D_i} \neg l)$*. Then,* $\psi \models \bigwedge_{i=1}^{m} (\bigvee_{l \in E_i} \neg l)$*.*

The above results are presented in terms of any implicants, rather than prime ones only. This is because the latter govern the rate of convergence, but irreducibility does not affect correctness. To conclude the elaborations, we observe that the greedy generation of prime implicants does not necessarily yield minimal formulae.

2.4 Experiments

We have implemented the techniques described in this chapter with the express aim of answering the following questions: (1) What is the overhead of using prime implicants compared to standard model enumeration? (2) What is the overhead of anytime quantifier elimination compared to the two-phase algorithm? (3) How are the implicants distributed in terms of size within the two phases of the algorithm? (4) How does the method compare against BDD-based projection, both in terms of the size of the resulting CNF formulae and the time required to produce them? To answer these questions, we compared our technique against a hybrid SAT/BDD approach. We implemented our methods on top of Sat4J and MiniSat v2.2. Cudd v2.4.2 was used for the BDD operations because it offers support for enumerating the

Table 2.1: Information regarding the benchmark set; column φ contains the name of the formula as referred to later on; the benchmarks at the bottom are generated from blocks of ATmega16 binary code; for these benchmarks, column *info* contains the number of instructions and whether they were generated for set abstraction (*set*) or transfer function synthesis (*tf*)

φ	info	$\lvert V\rvert/\lvert\varphi\rvert$	φ	info	$\lvert V\rvert/\lvert\varphi\rvert$
74181	74x series	1001/2368	s298	ISCAS-89	1327/3164
74182	74x series	227/526	s344	ISCAS-89	1665/3880
74283	74x series	267/646	s349	ISCAS-89	1678/3914
74L85	74x series	413/1084	s1196	ISCAS-89	5422/12870
add	3 (set)	74/119	adc	4 (tf)	19/290
increment	3 (set)	66/119	admdswp	11 (tf)	66/154
parity_mit	15 (set)	2066/6725	adsb2shad	8 (tf)	114/322
parity_swap	21 (set)	275/745	ilsh	5 (tf)	66/170
randerson	13 (set)	18658/61696	irsh	5 (tf)	66/170
triple_swap	9 (set)	89/192	iswp	8 (tf)	130/386

prime implicants of a BDD. We chose bitonic sorting for the sorting network, though smaller — albeit less regular — networks exist [146]. All experiments were performed on a 2.6 GHz MacBook Pro equipped with 4 GB of RAM.

2.4.1 Benchmarks

As benchmarks, we selected several circuits from the 74X and ISCAS-89 hardware benchmarks as well as projection problems arising from binary analysis (see Chap. 3 for details). The 74X circuits include an ALU (74181), a carry-look-ahead generator (74182), an adder (74283) and a magnitude comparator (74L85). The ALU is the hardest to analyze since it implements 16 different functions, depending on 4 input control bits. The ISCAS-89 benchmarks consist of a traffic light controller (s298), two 4×4 add-shift multipliers (s344 and s349), and a combinatorial circuit with randomly inserted flip-flops (s1196). All circuits were projected onto their inputs and outputs to express their semantics without reference to intermediate variables. The microcontroller code was exported from [MC]SQUARE [208] for the purpose of synthesizing transfer functions and for propagating ranges across blocks of ATMEL ATmega16 code. Table 2.1 presents the key statistics for each of these problems.

2.4.2 Anytime Quantifier Elimination

The results for some of the hardware benchmarks using SAT4J are given in Tab. 2.2. Here, it is important to appreciate that the projection of the 74185b formula does not

Table 2.2: Experimental results for projection using anytime quantifier elimination; here, column *#prime* refers to the maximum size of the computed implicants; column *#SAT* specifies the number of SAT instances to be solved, providing a hint on the number of spurious candidates

φ	$\lvert Y\rvert$	#prime	CNF	#SAT	time
74182	5	2	4	52	0.81
		5	4	170	1.80
74283	8	4	13	1590	5.63
		6	20	4053	14.49
		8	20	4881	16.71
74L85	10	4	6	4496	18.91
		5	14	12349	57.22
		6	30	24960	125.99
		8	30	47536	292.59
		10	30	51522	352.95

contain any implicants with size between 7 and 10. Likewise, 74283b does not have implicants of size 7 and 8. A similar distribution has been observed when widening is applied to Boolean formulae. Though the result of Kettle et al. [138] has not been obtained in the context of projection, it suggests that enumerating implicants up to a size threshold can achieve a good approximation of the projection. The ratio of the number of calls to the solver to the number of primes is largely due to spurious candidates (it roughly doubled by increasing the prime length by one or two), which motivates investigating the impact of instantiating variables. Clauses can be simplified after instantiation, which involves removing false literals from clauses and removing all clauses that were already satisfied. The effects of instantiation based on a model of the original formula are given in Tab. 2.3. The results in column *speedup* suggest that instantiation can significantly improve performance.

Finally, we study applying multiple instantiation, accompanied with simplification using straightforward constant propagation, for different instances of the 74L85b circuit. Note that simplification reduces the size of the SAT instance, which compensates somewhat for multiple instantiation. The instantiations themselves were generated from various models of the formula that were themselves found by applying blocking clauses. By choosing 6 instantiations that constrain the solution space in the 6/10 case a priori, the number of SAT instances reduced from 24960 to 16954, and the runtime decreased to 61.59s. This is a reduction of 32% in terms of the number of calls to a SAT solver and an overall speedup of 51%. Using 10 instantiations reduced the number of calls to the solver further to 14273 and took the runtime down to 52.45s, yielding a speedup of 58%. The key point is that

Table 2.3: Experimental results with a single instantiation

Formula	length	time	speedup
74182b	2/5	0.50s	38%
	5/5	0.85s	52%
74283b	4/8	4.26s	24%
	6/8	10.54s	27%
	8/8	12.34s	26%

Formula	length	time	speedup
74L85b	4/10	12.61s	23%
	5/10	38.85s	32%
	6/10	84.68s	33%
	8/10	203.45s	30%
	10/10	84.68s	33%

a reduction occurs in the ratio of the number of calls to the SAT solver and the number of primes. This is a measure of the effectiveness of the technique, i.e., how much effort is needed, on average, to find another implicant and thereby refine the approximation. However, we conjecture that it is not prudent to apply too many instantiations simultaneously, because at some point the size of the combined SAT instance will become unmanageable (this would correspond to a flattening of quantified bit-vector logic, which will eventually be prohibitively expensive).

2.4.3 Two-Phase Quantifier Elimination

Table 2.4 presents the results for DNF generation (respectively CNF conversion) using prime implicants, giving the number of implicants (respectively clauses) in the resulting formulae and the time required to compute them. Analogous figures are given for the hybrid approach. It is interesting to see that for the circuits s344 and s349, only 512 implicants in DNF are generated, but exhaustive model enumeration yields 65792 disjuncts. This is because 256 out of 512 implicants are of length 12, and thus already cover a large number of models in the projection space. This suggests that our method can make model enumeration tractable where the classical approach fails. For other cases, as exemplified by the 74181 and s1196 circuits, our approach offers no clear advantage. However, it is important to see that, for neither of the benchmarks, transformation seriously degrades performance; this is noteworthy because one cannot know the distribution of the primes a priori.

The percental distribution of the lengths of clauses that arise in CNF conversion and the evolution of the runtimes are depicted in Fig. 2.2. Graphs are given only for the 74X series, although these distributions appear to be typical. For DNF generation the distributions are less interesting, often consisting of a single spike, but sometimes consisting of two spikes, as for s344 and s349 at lengths 12 and 20. It is in these latter cases that primes improve over classical model enumeration.

Table 2.4: Experimental results for two-phase prime implicant enumeration with comparison to a hybrid method

φ	$\lvert Y\rvert$	Primes					Hybrid	
		DNF		CNF		total	size	total
		size	time	size	time	time		time
74181	22	16384	1.477	686	7.096	8.574	**476**	**2.798**
74182	13	320	0.025	26	0.009	**0.035**	**23**	0.039
74283	14	512	0.022	**98**	0.147	0.169	270	**0.099**
74L85	14	2048	0.108	**144**	0.107	0.215	145	**0.162**
s298	9	4	0.001	7	0.003	**0.004**	7	0.007
s344	20	512	0.068	16	0.018	**0.087**	16	0.098
s349	20	512	0.070	16	0.017	**0.088**	16	0.099
s1196	28	16384	11.182	**570**	5.465	**16.653**	822	16.993
adder	16	256	0.007	16	0.012	**0.020**	16	0.031
	24	1024	0.030	31	0.054	**0.086**	**29**	0.120
increment	8	4	0.001	10	0.001	**0.003**	10	0.007
	16	256	0.004	14	0.007	**0.012**	14	0.014
	24	256	0.008	**32**	0.024	**0.033**	34	0.035
parity_mit	8	100	0.033	4	0.001	**0.036**	4	0.039
	16	12800	2.363	16	1.361	3.727	**10**	**2.647**
	24	40960	8.543	**40**	6.316	14.875	41	**9.698**
parity_swap	8	16	0.002	4	0.001	**0.004**	4	0.009
	16	256	0.008	12	0.008	**0.017**	12	0.019
	24	256	0.013	**37**	0.038	**0.051**	40	0.114
randerson	8	64	0.102	2	0.001	**0.104**	2	0.108
	16	256	0.136	14	0.010	**0.147**	14	0.149
	24	256	0.140	**27**	0.023	**0.164**	30	0.198
triple_swap	8	16	0.002	12	0.001	**0.004**	12	0.009
	16	512	0.013	**20**	0.029	0.042	22	**0.028**
adc	8	128	0.004	7	0.002	**0.006**	7	0.010
	16	128	0.005	**47**	0.018	**0.023**	52	0.041
	24	128	0.006	**80**	0.047	**0.054**	92	0.122
admdswp	8	191	0.003	7	0.003	**0.006**	7	0.011
	16	191	0.007	**54**	0.021	**0.029**	60	0.057
	24	191	0.009	**56**	0.025	**0.035**	66	0.080
adsb2shad	16	154	0.008	**67**	0.026	**0.034**	71	0.097
	24	310	0.013	**124**	0.045	**0.058**	129	0.108

ilsh	8	32	0.002	3	0.001	**0.003**	3	0.010
	16	256	0.008	13	0.009	**0.017**	13	0.019
	24	256	0.009	**44**	0.023	**0.032**	46	0.046
irsh	8	16	0.001	4	0.001	**0.001**	4	0.008
	16	16	0.002	22	0.004	**0.006**	**21**	0.011
	24	16	0.003	45	0.012	**0.016**	45	0.020
iswp	16	4096	0.103	16	0.235	0.339	16	**0.179**
	24	4096	0.126	27	0.251	0.379	27	**0.266**
# Winner				**15**		**32**	4	8

2.5 Related Work

Over the past decades, quantifier elimination for Boolean logic (and other theories) has been a field of active research. We survey the most significant contributions to the field and reflect on the key differences compared to our techniques.

2.5.1 Consensus Method and Binary Resolution

A traditional technique to eliminate existential quantifiers from arbitrary Boolean formulae is Shannon expansion. To eliminate a variable x from φ, the original formula is instantiated twice and combined to give $\exists x : \varphi = \varphi[x \mapsto 0] \vee \varphi[x \mapsto 1]$. However, if φ is in CNF, then expansion does not preserve its structure. For a formula in CNF, it is well-known that existential quantifiers can be eliminated by repeatedly applying binary resolution. The advantage of this technique is that it takes as input a formula in CNF and results in a formula in CNF, too. To illustrate resolution, let

$$\varphi \quad = \quad (\bigwedge_{i=1}^{n_1}(x \vee C_i)) \wedge (\bigwedge_{j=1}^{n_2}(\neg x \vee D_j)) \wedge (\bigwedge_{k=1}^{n_3} E_k)$$

where C_i, D_j and E_k are clauses that involve neither x nor $\neg x$. Then, $\exists x : \varphi$ can be obtained by resolving each clause $x \vee C_i$ with each $\neg x \vee D_j$ to give:

$$\exists x : \varphi \quad = \quad (\bigwedge_{i=1}^{n_1}\bigwedge_{j=1}^{n_2}(C_i \vee D_j) \wedge (\bigwedge_{k=1}^{n_3} E_k)$$

This transformation increases the size of the representation by as many as $n_1 \cdot n_2 - n_1 - n_2$ clauses. Hence, the worst-case complexity is exponential [148]. For a propositional formula in DNF, the consensus method has independently been proposed by a number of researchers [30, 182, 205] as a way of enumerating all its prime implicants. If φ is in CNF, then it is straightforward to derive a DNF representation of $\neg\varphi$ by pushing negations inward. The consensus procedure can then be applied to $\neg\varphi$ to find its prime implicants. One might think that this

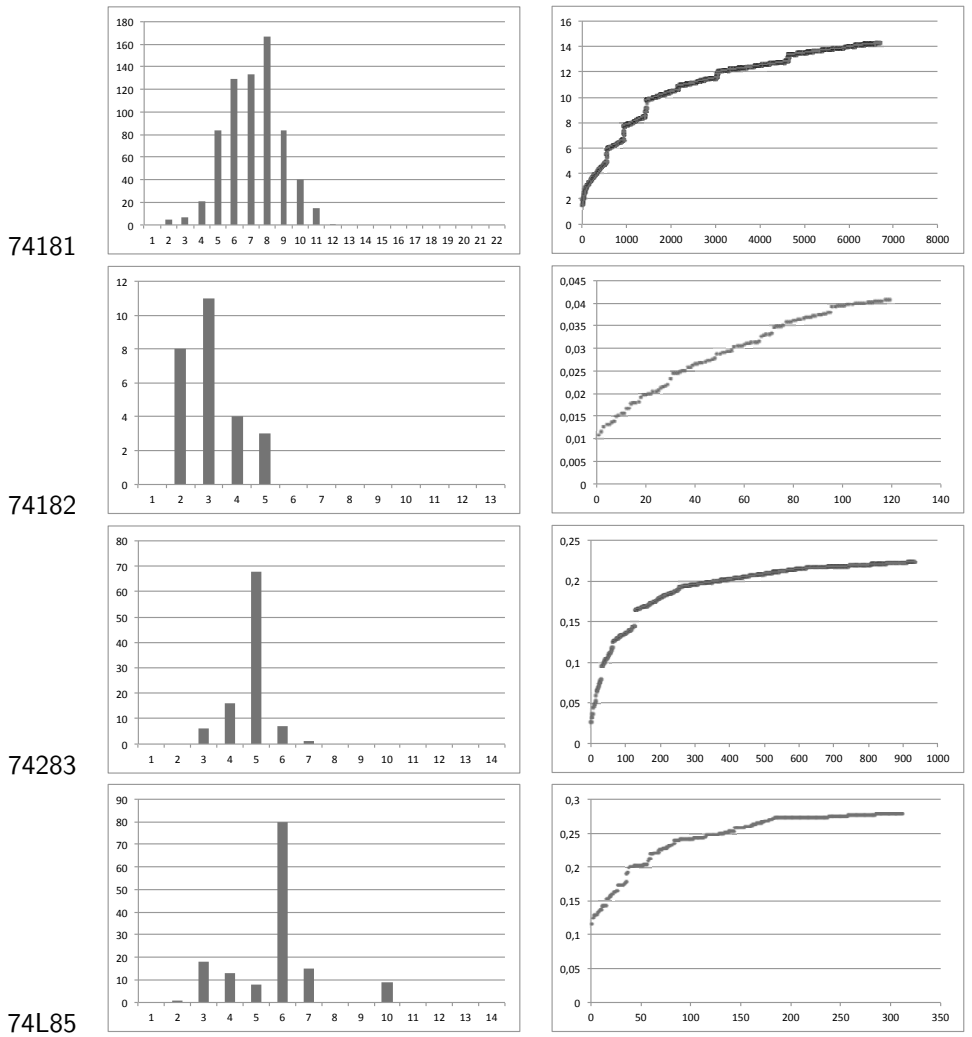


Figure 2.2: Distribution of implicants by length for the 74X benchmarks on the left; evolution of the runtimes for the overall number of computed implicants on the right (the x-axis contains the number of computed implicants, whereas the y-axis represents the runtime in seconds)

provides a way to compute projection, but the key step of the consensus method combines two elementary conjunctions of $\neg\varphi$, say, $(x) \wedge C$ and $(\neg x) \wedge D$, to form $C \wedge D$. This step is isomorphic to resolution. Hence, the consensus method shares the inefficiency problems associated with applying resolution to a formula in CNF.

2.5.2 Complexity of Prime Implicant Generation

The complexity of the shortest implicant problem for DNF formulae has been studied by Umans [236] who showed that it is $GC(\log_2(n), coNP)$-complete. Even though this result is not directly transferable to CNF, it suggests that the analogue problem in CNF may be similarly difficult, and thereby supports the application of SAT solvers to the derivation of shortest implicants. Integer linear programming has been used to find shortest implicants [177, 218], as have SAT engines which have been modified to support inequalities [163]. In [163], a transformation is described which is similar to $\tau(\psi)$, but the work is not concerned with quantifier elimination. Hence, binary variables are introduced for each variable in the formula rather than merely those in the projection space. Techniques akin to the τ transformation are known as *dual-rail encodings* in the community, referring to the duality of encoding positive and negative results in a single formula [48].

2.5.3 Hybrid Methods and McMillan's Method

SAT solving has been used before to compute projections [152, 164] as have BDDs [52, 152, 240]. Apart from these techniques which focus on one way of representing a Boolean formula, hybrid approaches exist which combine SAT solving and BDDs. Such hybrid approaches typically represent state sets as BDDs and express the transition relation in CNF, as done by Gupta et al. [126] and later by Sheng and Hsiao [217]. Yet, some approaches combine BDDs and SAT solving in different ways. For example, Damiano and Kukula [88] replace clauses with BDDs in their SAT solver GRASP. In particular, they allow a clause to be substituted by any Boolean formula represented as a BDD, so that the problem to be solved consists of a conjunction of BDDs. Jin and Somenzi [131] combine BDDs and SAT solving using CNF to avoid explosion in the sizes of the resulting BDDs. In their tool CIRCUS, they determine clusters of clauses, which are then grouped into BDDs and converted back into clauses (under certain conditions). In a different context, Aloul et al. [1] have studied the connection between CNF formulae and BDDs for efficient variable orderings. The approach of Cavada et al. [56] provides a technique for recursive quantification. They apply this technique to subtrees, which are then combined. SMT solving is used to ensure consistency of the transformations applied.

McMillan [164] has shown how to perform universal projection for CTL modalities such as $\mathsf{AX}\varphi$ using DPLL-like enumeration and also explained how to represent an arbitrary Boolean encoding of φ in CNF without existentially quantified variables. The key idea of his `toCNF(`φ`)` procedure [164, Sect. 3] is to deduce a clause from a satisfying assignment of φ whose complement rules out some cases that violate φ. His approach requires a modified DPLL-engine and resolution coupled with several heuristics — which literals to analyze, which variables to resolve on and

suchlike — which strongly affect the performance of the approach [164, Sect. 2]. Our approach, in comparison, builds on top of an existing SAT library and is therefore both straightforward to implement and will immediately benefit from any improvement to the library itself. Nevertheless, we consider the SAT-based algorithm of McMillan to be an important work that has indeed found a wide range of applications beyond SAT-based model checking. To consider only some applications, Clarke et al. [68] integrated the technique into predicate abstraction of hardware circuits, whereas Chauhan et al. [59] used it for post-image computation. A variation on the McMillan algorithm is given by Sheng and Hsiao [217] who apply a success-driven rather than a conflict-driven search for models (recall that DPLL-style algorithms use a conflict-driven search). However, Sheng and Hsiao store their results in a BDD rather than generating a CNF formula.

2.5.4 Methods based on Prime Implicants and Cubes

Prime implicants have been directly applied to widening Boolean functions represented as ROBDDs [138]. By applying a recursive meta-product construction [76], collections of short prime implicants can be used to derive an ROBDD that is an over-approximation of the input. Our work on applying SAT to projection was motivated by the empirical finding that collections of short primes often yield good approximations of Boolean formulae [138, Sect. 5.1].

Lahiri et al. [152] have described how to enumerate cubes in the projection space using SAT solving so as to perform image computation for predicate abstraction. Blocking clauses are chosen heuristically — though details of the heuristics are not given — and the approach does not guarantee to infer cubes of minimal size. In their experimental evaluation, the authors compare a SAT- against a BDD-based elimination scheme and observe that the efficiency of either method is sensitive to the number of variables projected onto (which has later been noted by Bryant [50], too). This work was further developed by Lahiri et al. [153] who used DPLL(T)-based SMT solving to enumerate models. Each model is then stored in a BDD from which the results are extracted as a disjunction of prime implicants. They search for cubes $c_{1,k}, \ldots, c_{n,k}$ of increasing length k such that $\varphi \models \bigvee_{i=1}^{n} c_{i,k}$, which chimes with our approach. By way of contrast, however, we apply prime implicants in two different ways, namely, for enumerating cubes as well as clauses, so that our final quantifier-free formula is presented in CNF. We shall discuss the conceptual difference between the algorithm of Lahiri et al. and our method by means of an example. Suppose that model enumeration using our technique yields 256 cubes of lengths 12 and 20. The method of Lahiri et al. enumerates all intermediate cubes of lengths $13, \ldots, 19$ to converge onto $\exists X : \varphi$, whereas our approach leapfrogs these intermediate cubes by specifying the requirement of a cube of size k within the SAT instance itself. Earlier approaches [89, 99, 204] to predicate abstraction invoke a

SAT solver for each cube c to discover if $\varphi \wedge c$ is satisfiable. To reduce the number of calls to the decision procedure, these techniques start with small cubes and — only if $\varphi \wedge c$ is satisfiable — proceed with cubes of the form $\varphi \wedge c \wedge d$ and $\varphi \wedge c \wedge \neg d$. This approach is based on a large number of SAT/SMT calls, typically requires many unsatisfiability proofs (which are often more difficult for SAT solvers to provide than to find a model), and does not fit as well with incremental SAT [153, Sect. 3.2].

More recently, Monniaux [168] described a method for quantifier elimination called lazy model enumeration. The key idea of his algorithm is to derive a cube c that implies a given formula $\exists X : \varphi$, which is then generalized towards a weaker (more general) implicant c' such that $c \models c' \models \exists X : \varphi$. By way of comparison, our algorithm starts with implicants as short as possible. The relaxation step in our method is required by the design of the blocking clauses, which suppress more implicants than those derived last. Although similar in spirit, his algorithm proceeds diametrically opposed to ours, and moreover generates DNF. The MathSat SMT solver [46] uses an algorithm that also relies on a formula transformation similar to τ. However, rather than adding cardinality constraints to the SAT instance, they modified the solver so that it prefers 0-decisions during SAT solving.

2.5.5 Methods for Quantified Boolean Formulae

Our techniques are, of course, related to quantified Boolean formula (QBF) solving [55, 179], too. Most contributions to the field of QBF solving, however, deal with alternating quantifiers, whereas our approach handles existential quantification only. A recent contribution to QBF solving for bit-vector theories was described by Wintersteiger et al. [243]. Most notably, they integrate heuristics based on word-level simplifications and templates so as to ease the structure of the problem. We believe that a combination of word-level heuristics with our elimination algorithm (when applied to bit-vectors) appears promising, too.

2.6 Discussion

This chapter has discussed two different strategies for eliminating variables from propositional Boolean formulae using existential projection. Both techniques rely entirely on off-the-shelf SAT engines and a dual-rail encoding that specifies shortest prime implicants within the SAT instance itself. We have identified two distinct properties of these algorithms, the first being *interruptible*, whereas the second one exhibits the *everyone a winner* property. Although the algorithm from Chap. 2.3 is drastically more efficient than the one presented in Chap. 2.2, there are certain situations where the latter shows to be useful: even if computing the overall projection is intractable, over-approximate results may valuable [137, 138].

3 Control Flow Reconstruction using Boolean Logic

From existential quantifier elimination, we now turn to an application in binary code analysis. When analyzing binaries, one particular challenge is posed by indirect control instructions [20, 61, 91, 100, 129, 140–142, 187]. Such instructions pass control to another instruction that is determined by a concrete value held in some register. Control flow is thus computed at runtime, and may change in any execution of the program. This situation poses a so-called chicken-and-egg problem [142, Sect. 1]: In order to reconstruct a precise over-approximation of the control flow graph (CFG) from unstructured binary code, it is necessary to compute precise invariants that describe those registers which affect the target of an indirect jump. A CFG is, in turn, required to compute these invariants. CFG reconstruction is thus a key problem in binary code analysis: If the reconstruction is unsafe, then any subsequent analysis is unsafe; if it is overly conservative, the analysis results are overly conservative, too.

Motivating Example In presence of indirect control, the lack of an accurate CFG often implies a drastic loss in terms of precision for any subsequent verification effort, caused by spurious edges in the CFG. To illustrate, we discuss the loss in precision incurred by spurious jump targets by means of an example:

```
#define SWP(a,b) (a^=b, b^=a, a^=b, a&=0x0F, b&=0x0F)
```

The macro `SWP(a,b)` swaps the contents of two variables `a` and `b` without involving a third, based on three exclusive-or operations. The last two operations clear the upper nibbles of the results. Indeed, the above macro implements a well-known idiom [239, Chap. 2.19]. For the accumulator-based Intel MCS-51 microcontroller, the macro is compiled into seven instructions as depicted on the left. Now assume the `SWP(a,b)` macro is used within a switch-case statement as given in Fig. 3.1. For the Intel MCS-51, the switch-case statement is compiled into a jump table, which is stored in program memory. Upon reaching the switch-case statement, the application looks up a comparison value and a jump

```
MOV A, 0x05;
XRL A, 0x25;
XRL 0x25, A;
XRL A, 0x25;
MOV 0x05, A;
ANL 0x25, #0x0F;
ANL 0x05, #0x0F;
```

```
1 switch (p) {
2         case 10: SWP(x,y); break;
3         case 20: foo(x,y); break;
4         default: bar(x,y)
5 }
```

Figure 3.1: Program fragment which is compiled into a jump table

target `pc` from program memory; these two values constitute an entry in the jump table. Address `pc` indicates the first instruction of the respective `case` block. If the comparison matches, control is redirected to `pc`, i.e., the program counter is assigned `pc`. In the example, the jump table consists of three comparison values 10, 20 and #`default`. The jump targets are given by addresses `0x100` for the `SWP` macro and two addresses `0x110` and `0x118` for the calls of `foo` and `bar`.

Junk Instructions in Control Flow Recovery Now assume that an abstract interpreter computes a range $I = [\texttt{0x100}, \texttt{0x110}]$ for the jump target `pc` using interval analysis [40, 186]. On architectures with instructions of variable length — typically CISC — edges need to be added to the CFG at the granularity of the shortest possible instruction length, i.e., byte-level granularity for the Intel MCS-51. The need to soundly approximate the CFG entails that edges from the indirect jump to all concretizations of I, i.e., $\texttt{0x100}, \ldots, \texttt{0x110}$ are added to the CFG. The first value `0x100` points to the first instruction `MOV A, 0x05` of the `SWP` macro, represented by a two-byte opcode `0xE5:0x05`. The address `0x102` indicates the instruction `XLR A, 0x25`, which corresponds to the opcode `0x65:0x25`. However, we also have $\texttt{0x101} \in I$, and the second byte of `MOV A, 0x05` paired with the first byte of `XLR A, 0x25` forms a new instruction `INC 0x65`, which increments memory location `0x65` by one. Likewise, all addresses in I indicate valid sequences of instructions:

`0x100`	=	`0xE5:0x05`	↦	`MOV A, 0x05`	legitimate
`0x101`	=	`0x05:0x65`	↦	`INC 0x65`	spurious
`0x102`	=	`0x65:0x25`	↦	`XRL A, 0x25`	legitimate
`0x103`	=	`0x25:0x62`	↦	`ADD A, 0x62`	spurious
`0x104`	=	`0x62:0x25`	↦	`XRL, 0x25, A`	legitimate
`0x105`	=	`0x25:0xF5`	↦	`ADD A, 0x65`	spurious
`0x106`	=	...	↦	...	legitimate

The jump target `0x100` thus induces a basic block

```
0x100:  MOV A, 0x05;    0x102:  XRL A, 0x25;    ...
```

whereas the jump target `0x101` indicates a different block:

```
0x101:  INC 0x65;    0x103:  ADD A, 0x62;    ...
```

We refer to fragments in the executable such as the block induced by address `0x101` as *junk code*: it consists of opcodes that are never executed, but only form part of the program due to over-approximation of the analysis. Subsequent analyses, which are based on the disassembled binary, will therefore calculate invariants using junk instructions, too. This leads to a significant loss in precision — also referred to as *unbearable noise propagation* in the literature [20, Sect. 1] — which inevitably leads to large numbers of spurious warnings. The situation is even worsened if the junk code coincides with indirect or (un-)conditional jump instructions, thereby adding further control flow, though this is not the case in the above example. Albeit the derived interval appears to be tight at a first glance since its boundaries coincide with actual jump targets, the choice of abstract domain inevitably induces an imprecise reconstruction of the CFG. Further, there is no reason why jump targets should be distributed along a regular pattern, which suggests that value sets rather than (strided) intervals form an appropriate abstract domain for control flow analysis.

Yet, indirect control is ubiquitous in compiler-generated as well as in handcrafted assembly code. Apart from `switch` statements, compilers generate indirect jumps or calls for function pointers or virtual method calls. A more subtle case in point is given for `return` statements, which alter the program counter according to a value that has been stored on the runtime stack. In certain situations, compilers intentionally alter the runtime stack, e.g., when evaluating jump tables. The generated code then exhibits much similarity with code that exploits possibilities for buffer overruns. It is thus hardly possible to overestimate the value of precise invariants in order to verify safety properties of binaries.

Contributions and Outline To summarize our work, this chapter contributes a fully automatic algorithm for control flow reconstruction using Boolean logic. The technique itself only requires a symbolic relational encoding of the instruction-set semantics of the target hardware. The encodings are then combined towards a formula that represents the semantics of a basic block (a sequence of instructions) as a whole. Based on these encodings, forward and backward value set analyses are straightforwardly implemented using existential quantification combined with a dedicated abstraction procedure [21]. As shown in Chap. 2.3, quantification can efficiently be implemented as incremental SAT, and so can value set abstraction. By resting the analysis on a relational encoding of the instruction set, forward and backward analyses can then be executed uniformly. The algorithm itself exhibits several interesting properties:

1. It is sound in the sense that it does not miss any edges in the CFG.

2. It turns out to be exact for many examples of typical microcontroller programs.
3. It is generic, although we demonstrate and evaluate it for the Intel MCS-51.

The remainder of this chapter builds towards these contributions as follows. First, Chap. 3.1 presents an algorithm that performs block-wise value set abstraction. Given a basic block, the algorithm relies on a propositional encoding of the entire block to compute value sets of registers that are accessed within the block. Then, Chap. 3.2 lifts the approach to entire programs, which merely amounts to a forward fixed-point computation in the abstract interpretation framework, which is interleaved with depth-bounded refinements in backward direction. Experimental results are presented in Chap. 3.3, before we conclude with a survey of related work in Chap. 3.4 and a discussion in Chap. 3.5.

3.1 Block-Level Abstraction

Our technique for control flow recovery is based on the idea of deriving invariants in terms of pre- and postconditions for all blocks in a program. Pre- and postconditions are themselves expressed as conjunctions of value sets, which appears a natural choice since there is no reason why jump targets should be distributed along a predictable pattern (cp. [20, Sect. 1]). The algorithm to compute value sets on input and output of a basic block is, in turn, based on an application of the projection scheme introduced in Chap. 2.3. The relation between these key components is discussed in the following.

3.1.1 Bit-Blasting Blocks

To illustrate the representation of programs as bit-vector relations, consider the exclusive-or operations used within the `SWP` macro, i.e., `XLR a, b` where `a` and `b` are registers. This instruction computes the bitwise exclusive-or of `a` and `b` and stores the result in `a`. To express its semantics, we introduce two bit-vectors $\boldsymbol{a} = (\boldsymbol{a}[0], \ldots, \boldsymbol{a}[7])$ and $\boldsymbol{b} = (\boldsymbol{b}[0], \ldots, \boldsymbol{b}[7])$ to represent the values of `a` and `b` on entry of the instruction. Likewise, we introduce a bit-vector $\boldsymbol{a}'$ to represent the value of `a` on exit (`b` remains unchanged). With $\oplus$ denoting the Boolean exclusive-or, the semantics of `XLR a, b` is then expressed using a Boolean formula as follows:

$$[\![\texttt{XLR a, b}]\!] \;=\; \textstyle\bigwedge_{i=0}^{7} (\boldsymbol{a}'[i] \leftrightarrow (\boldsymbol{a}[i] \oplus \boldsymbol{b}[i]))$$

The force of such encodings is that they can be used to reason about executions of the encoded instruction (or block) in both, forward and backward direction. Thus, given some predicate $\xi(\boldsymbol{a}')$ that constrains $\boldsymbol{a}'$ on exit, the conjoined formula

$$[\![\texttt{XLR a, b}]\!] \;\wedge\; \xi(\boldsymbol{a}')$$

implicitly describes combinations of $\boldsymbol{a}$ and $\boldsymbol{b}$ on entry that satisfy the predicate $\xi(\boldsymbol{a}')$ on $\boldsymbol{a}'$. Similarly, given a predicate $\xi(\boldsymbol{a})$ on $\boldsymbol{a}$ on entry, it is also possible to invoke a SAT solver to infer combinations of values of $\boldsymbol{b}$ and $\boldsymbol{a}'$ that respect $\xi(\boldsymbol{a})$. As another example, consider the instruction `INC a` found in the junk code mentioned in the introduction. This instruction is encoded over bit-vectors $\boldsymbol{a}$ and $\boldsymbol{a}'$:

$$[\![\texttt{INC a}]\!] \;=\; \textstyle\bigwedge_{i=0}^{7} \left(\boldsymbol{a}'[i] \leftrightarrow \boldsymbol{a}[i] \oplus \bigwedge_{j=0}^{i-1} \boldsymbol{a}[j] \right)$$

In the same spirit, encodings for the entire instruction set can be derived [39]. Given a set $\boldsymbol{V} = \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\}$ of bit-vectors $\boldsymbol{v}_i = (\boldsymbol{v}_i[0], \ldots, \boldsymbol{v}_i[w-1])$, we interpret $\varphi \in \wp(\wp(\mathbb{B}^{n \times w}))$ as a formula in CNF. Thus, with $\boldsymbol{V} = \{\boldsymbol{v}\}$, a formula such as $(\boldsymbol{v}[0] \vee \boldsymbol{v}[1]) \wedge (\boldsymbol{v}[2])$ is represented as $\{\{\boldsymbol{v}[0], \boldsymbol{v}[1]\}, \{\boldsymbol{v}[2]\}\} \in \wp(\wp(\boldsymbol{V}))$.

Definition 3.1. *Denote the encoding of an instruction* `op` *over bit-vectors* $\boldsymbol{V}$ *in propositional Boolean logic by* $[\![\texttt{op}]\!] \in \wp(\wp(\boldsymbol{V}))$.

Given a block $b = (\texttt{op}_1, \ldots, \texttt{op}_m)$ of m instructions, the map $[\![\cdot]\!]$ can straightforwardly be lifted from instructions to blocks: (1) apply static single assignment (SSA) conversion [87] to avoid accidental coupling if registers are accessed in b more than once, and (2) compute $[\![b]\!]$ by conjoining the constituent instructions.

Example 3.1. *We illustrate this concept for a block b that corresponds to the* `SWP` *macro from the introduction. We represent memory locations* `0x05` *and* `0x25` *using bit-vectors* $\boldsymbol{x}$ *and* $\boldsymbol{y}$*, whereas the accumulator* `A` *is represented by* $\boldsymbol{a}$*. Then, SSA conversion leads to fresh bit-vectors* $\boldsymbol{a}', \boldsymbol{a}'', \boldsymbol{x}', \boldsymbol{x}'', \boldsymbol{y}'$*, and* $\boldsymbol{y}''$*, the relations among which a formula* $[\![b]\!] \in \wp(\wp(\{\boldsymbol{a}, \boldsymbol{a}', \boldsymbol{a}'', \boldsymbol{x}, \boldsymbol{x}', \boldsymbol{x}'', \boldsymbol{y}, \boldsymbol{y}', \boldsymbol{y}''\}))$ *equivalently describes:*

$$[\![b]\!] \;=\; \begin{cases} (\bigwedge_{i=0}^{7} \boldsymbol{a}[i] \leftrightarrow \boldsymbol{x}[i]) & \wedge \;\; (\bigwedge_{i=0}^{7} \boldsymbol{a}'[i] \leftrightarrow (\boldsymbol{a}[i] \oplus \boldsymbol{y}[i])) \;\; \wedge \\ (\bigwedge_{i=0}^{7} \boldsymbol{y}'[i] \leftrightarrow \boldsymbol{y}[i] \oplus \boldsymbol{a}'[i]) & \wedge \;\; (\bigwedge_{i=0}^{7} \boldsymbol{a}''[i] \leftrightarrow \boldsymbol{a}'[i] \oplus \boldsymbol{y}'[i]) \;\; \wedge \\ (\bigwedge_{i=0}^{7} \boldsymbol{x}'[i] \leftrightarrow \boldsymbol{a}''[i]) & \wedge \;\; (\bigwedge_{i=4}^{7} \boldsymbol{y}''[i] \leftrightarrow \neg \boldsymbol{y}'[i]) \;\; \wedge \\ (\bigwedge_{i=4}^{7} \boldsymbol{x}''[i] \leftrightarrow \neg \boldsymbol{x}'[i]) & \end{cases}$$

$\boldsymbol{x}''$ *and* $\boldsymbol{y}''$ *represent the values of* `0x05` *and* `0x25` *on output.*

Passing a formula to a solver necessitates CNF conversion, which can be achieved using flattening. In our implementation, we apply Tseitin's algorithm [178, 234] to avoid exponential growth of $[\![b]\!]$ [165], introducing existentially quantified variables $\boldsymbol{T}$. We should thus denote the resulting CNF formula for a block b by $[\![b]\!] \in \wp(\wp(\boldsymbol{V} \cup \boldsymbol{T}))$. However, we omit this detail for the purpose of presentation.

3.1.2 Value Set Abstraction using Incremental SAT Solving

To compute value sets of registers described by a formula φ, we apply an iterative algorithm [21, Sect. 3] that derives value set abstractions for bit-vectors. We define:

Definition 3.2. *Let* $\langle\cdot\rangle : \boldsymbol{V} \to \mathbb{N}$ *defined as* $\langle \boldsymbol{v} \rangle = \sum_{i=0}^{w-1} 2^i \cdot \boldsymbol{v}[i]$ *denote the unsigned value of a bit-vector* $\boldsymbol{v} = (\boldsymbol{v}[0], \ldots, \boldsymbol{v}[w-1])$*. Likewise, let* $\langle\langle\cdot\rangle\rangle : \boldsymbol{V} \to \mathbb{Z}$ *defined as* $\langle\langle \boldsymbol{v} \rangle\rangle = -2^{w-1} \cdot \boldsymbol{v}[i] + \sum_{i=0}^{w-2} 2^i \cdot \boldsymbol{v}[i]$ *denote the signed interpretation of* $\boldsymbol{v}$*.*

Corollary 3.1. *Let* $\boldsymbol{v} = (\boldsymbol{v}[0], \ldots, \boldsymbol{v}[w-1])$*. Then:*

- $\langle \boldsymbol{v} \rangle \in \{0, \ldots, 2^w - 1\}$
- $\langle\langle \boldsymbol{v} \rangle\rangle \in \{-2^{w-1}, \ldots 2^{w-1} - 1\}$

To converge onto the value sets of $\boldsymbol{v}$, the key idea of the algorithm is to alternately compute interval abstractions of φ and $\neg\varphi$. In what follows, we describe a well-known algorithm to derive intervals (cp. [21, Sect. 2], [39, Sect. 6] and [71, Sect. 3]) and then discuss how interval abstraction is applied during value set analysis.

Interval Abstraction

Before discussing how to compute interval abstractions of bit-vectors, we define the underlying abstract domain of intervals formally:

Definition 3.3. *We define the interval abstract domain as the complete lattice* $(\mathit{Int}, \sqsubseteq_{\mathit{int}})$ *with*

$$\mathit{Int} \quad = \quad \{[\ell, u] \mid 0 \leq \ell \leq u \leq 2^w - 1\} \quad \cup \quad \{\bot_{\mathit{int}}\}$$

where $\bot_{\mathit{int}}$ *denotes the empty interval. Naturally, we have* $\top_{\mathit{int}} = [0, 2^{w-1}]$*. The partial order* $\sqsubseteq_{\mathit{int}}$ *is induced by the subset relation.*

A procedure $\texttt{maximum} : (\wp(\wp(\boldsymbol{V})) \times \boldsymbol{V}) \to \mathbb{N}$ to compute the value of a concrete bit-vector $\boldsymbol{k}$ that represents the least upper bound of $\boldsymbol{v}$ in the interval domain Int is given in Alg. 1. The key idea of this algorithm is to instantiate single bits of $\boldsymbol{v}$ with true, starting with the most significant bit, and to repeatedly test satisfiability. Then, for instance, satisfiability of a formula $\varphi \wedge \boldsymbol{v}[7]$ corresponds to the existence of a model of φ such that $\langle \boldsymbol{v} \rangle \geq 128$. If satisfiable, the least upper bound $\langle \boldsymbol{k} \rangle$ of $\langle \boldsymbol{v} \rangle$ is found in the interval $[128, 255]$, and in $[0, 127]$ otherwise. By instantiating the remaining bits of $\boldsymbol{v}$ one after another, the interval that contains $\langle \boldsymbol{k} \rangle$ is incrementally refined using binary search. Given a bit-vector $\boldsymbol{v} = (\boldsymbol{v}[0], \ldots, \boldsymbol{v}[w-1])$ of length w, this strategy requires w calls to a SAT solver to compute $\langle \boldsymbol{k} \rangle$ exactly since Boolean formulae are discrete.

Proposition 3.1. *Alg. 1 computes the least upper bound of* $\langle \boldsymbol{v} \rangle$ *subject to* φ*, formally* $\mathit{maximum}(\varphi, \boldsymbol{v}) = \max\{u \mid u \in [0, 2^{w-1}]$ *and* $\varphi \wedge (\langle \boldsymbol{v} \rangle = u)$ *is satisfiable*$\}$*.*

Algorithm 1 $\texttt{maximum} : (\wp(\wp(\boldsymbol{V})) \times \boldsymbol{V}) \to \mathbb{N}$

Input: formula $\varphi \in \wp(\wp(\boldsymbol{V}))$
Input: bit-vector $\boldsymbol{v} = (\boldsymbol{v}[0], \ldots, \boldsymbol{v}[w-1]) \in \boldsymbol{V}$
Output: least upper bound $\langle \boldsymbol{k} \rangle \in \mathbb{N}$ of $\boldsymbol{v}$ subject to φ

```
1:  k ← ⟨⟩
2:  while |k| < w do
3:    if φ ∧ v[w − 1 − |k|] is satisfiable then
4:      φ ← φ ∧ v[w − 1 − |k|]
5:      k ← ⟨1⟩ : k
6:    else
7:      φ ← φ ∧ ¬v[w − 1 − |k|]
8:      k ← ⟨0⟩ : k
9:    end if
10: end while
11: return ⟨k⟩
```

The algorithm is adapted to greatest lower bounds of $\langle \boldsymbol{v} \rangle$ as follows. Swap the occurrences of $\boldsymbol{v}[w-1-|\boldsymbol{k}|]$ and $\neg\boldsymbol{v}[w-1-|\boldsymbol{k}|]$, and likewise swap (1) and (0) in lines 5 and 8 of Alg. 1. We denote this modified procedure $\texttt{minimum} : (\wp(\wp(\boldsymbol{V})) \times \boldsymbol{V}) \to \mathbb{N}$. Both procedures can be adapted to compute extremal values of signed bit-vectors, too; in that case, the most significant bit, which represents the sign, needs to be handled properly (cp. [21, Sect. 3]).

Definition 3.4. *Let $\varphi \in \wp(\wp(\boldsymbol{V}))$ denote a formula over $\boldsymbol{V} = \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\}$. We define the interval abstraction $\alpha_{\mathit{int}} : (\wp(\wp(\boldsymbol{V})) \times \boldsymbol{V}) \to \mathit{Int}$ of $\boldsymbol{v} \in \boldsymbol{V}$ subject to φ as:*

$$\alpha_{\mathit{int}}(\varphi, \boldsymbol{v}) \quad = \quad [\mathit{minimum}(\varphi, \boldsymbol{v}), \mathit{maximum}(\varphi, \boldsymbol{v})]$$

We present an example to highlight the internal steps of α_{int}.

Example 3.2. *Consider φ over $\boldsymbol{y} = (\boldsymbol{y}[0], \ldots, \boldsymbol{y}[5])$ defined as:*

$$\varphi \quad = \quad \begin{cases} (\neg\boldsymbol{y}[1] \vee \boldsymbol{y}[5]) \wedge (\neg\boldsymbol{y}[2] \vee \neg\boldsymbol{y}[5]) \wedge (\boldsymbol{y}[4] \vee \boldsymbol{y}[5]) \wedge (\boldsymbol{y}[2] \vee \neg\boldsymbol{y}[4]) \wedge \\ (\boldsymbol{y}[3] \vee \neg\boldsymbol{y}[5]) \wedge (\boldsymbol{y}[0] \vee \boldsymbol{y}[5]) \wedge (\neg\boldsymbol{y}[0] \vee \neg\boldsymbol{y}[1]) \wedge (\neg\boldsymbol{y}[3] \vee \boldsymbol{y}[5]) \end{cases}$$

Clearly $\langle \boldsymbol{y} \rangle \in [0, 63]$, and hence $\langle \boldsymbol{k} \rangle \in [0, 63]$. To find a least upper bound $\langle \boldsymbol{k} \rangle$ of $\langle \boldsymbol{y} \rangle$ subject to φ, we apply Alg. 1. In the first iteration, we test $\varphi \wedge \boldsymbol{y}[5]$ for satisfiability, which corresponds to searching for a model of φ such that $\langle \boldsymbol{y} \rangle \geq 32$; since satisfiable, we deduce $\langle \boldsymbol{k} \rangle \in [32, 63]$. In the second iteration, we test $\varphi \wedge \boldsymbol{y}[5] \wedge \boldsymbol{y}[4]$ for satisfiability. From unsatisfiability, we infer a range $[32, 47]$ for $\langle \boldsymbol{k} \rangle$. After four more iterations, we find the concrete value $\langle \boldsymbol{k} \rangle = 2^5 + 2^3 + 2^1 = 42$. The overall progress of the algorithm applied to φ is highlighted in Fig. 3.2. Applying the converse algorithm to compute the minimum of $\langle \boldsymbol{y} \rangle$ yields 22, which entails $\alpha_{\mathit{int}}(\varphi, \boldsymbol{y}) = [22, 42]$.

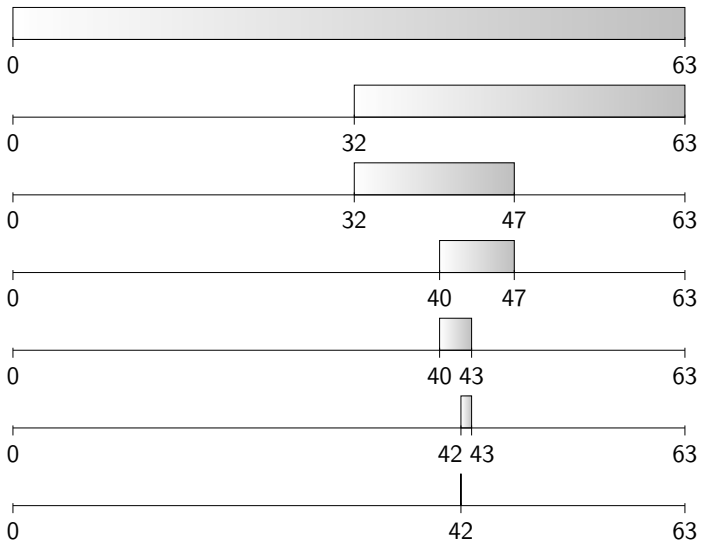


Figure 3.2: Progress of least upper computation in Ex. 3.2

Value Set Abstraction

The (unsigned) value set domain consists of (possibly non-contiguous) subsets of $\{0, \ldots, 2^w - 1\}$ for each bit-vector $\boldsymbol{v}$, with a partial order induced by the $\subseteq$-relation.

Definition 3.5. *Let* $\mathbb{Z}_w = \{0, \ldots, 2^w - 1\}$. *Then,* $(\mathsf{Val}, \sqsubseteq_{\mathsf{val}})$ *with* $\mathsf{Val} = \wp(\mathbb{Z}_w)$ *and* $a \sqsubseteq_{\mathsf{val}} b \Leftrightarrow a \subseteq b$ *is called the* value set abstract domain.

The join $\sqcup_{\mathsf{val}} : (\mathsf{Val} \times \mathsf{Val}) \to \mathsf{Val}$ of two value sets can straightforwardly be defined as the union of sets, likewise for the meet $\sqcap_{\mathsf{val}} : (\mathsf{Val} \times \mathsf{Val}) \to \mathsf{Val}$. A procedure to compute value sets of $\boldsymbol{v}$ rather than intervals is given in Alg. 2. The procedure takes as input a formula $\varphi \in \wp(\wp(\boldsymbol{V}))$ and a bit-vector $\boldsymbol{v} \in \boldsymbol{V}$. As a preprocessing step, φ is projected on $\boldsymbol{v}$ using existential quantifier elimination (line 3 in Alg. 2). Projection thus yields a formula $\psi \in \wp(\wp(\{\boldsymbol{v}\}))$. Then, to compute the value sets of $\boldsymbol{v}$, the algorithm alternates between over- and under-approximations. First, lower and upper bounds ℓ and u for $\langle \boldsymbol{v} \rangle$ are computed using minimization and maximization as given in Alg. 1, and we set $S = \{\ell, \ldots, u\}$. In a second iteration, an over-approximate range of $\langle \boldsymbol{v} \rangle$ described by $\neg\psi$ is computed; this range is removed from S. The third iteration again extends S, etc. until the result stabilizes.

Proposition 3.2. *Alg. 2 computes the least value set abstraction of* $\langle \boldsymbol{v} \rangle$ *subject to* φ, *i.e.,* $\alpha_{\mathsf{val}}(\varphi, \boldsymbol{v}) = \{v \mid v \in [0, 2^w - 1]$ *and* $\varphi \wedge (\langle \boldsymbol{v} \rangle = v)$ *is satisfiable*$\}$.

Algorithm 2 $\alpha_{\mathsf{val}} : (\wp(\wp(\boldsymbol{V})) \times \boldsymbol{V}) \to \mathsf{Val}$

```
1:  S ← ∅
2:  p ← true
3:  ψ ← project(φ, v)
4:  ℓ ← 0
5:  u ← 2^w − 1
6:  while ⟨l⟩ < ⟨u⟩ do
7:     ℓ ← minimum(ψ ∧ (ℓ ≤ ⟨v⟩), v)
8:     u ← maximum(ψ ∧ (⟨v⟩ ≤ u), v)
9:     if p then
10:       S ← S ∪ {ℓ, ... u}
11:    else
12:       S ← S \ {ℓ, ... u}
13:    end if
14:    ψ ← ¬ψ
15:    p ← ¬p
16: end while
17: return S
```

The difference of Alg. 2 compared to [21, Sect. 3] is found in line 3. Here, φ is projected onto $\boldsymbol{v}$ to give ψ. To illustrate the difference, let $\widehat{\boldsymbol{V}} = \boldsymbol{V} \setminus \{\boldsymbol{v}\}$. In general, c such that $c \models (\exists \widehat{\boldsymbol{V}} : \varphi \wedge \exists \widehat{\boldsymbol{V}} : \neg\varphi)$ may exist, hence the need to operate on $\exists \widehat{\boldsymbol{V}} : \varphi$ and $\neg(\exists \widehat{\boldsymbol{V}} : \varphi)$ since $c \models \exists \widehat{\boldsymbol{V}} : \varphi$ iff $c \not\models \neg\exists \widehat{\boldsymbol{V}} : \varphi$. Projection ensures progress of Alg. 2 in each iteration, and thus sidesteps the requirement $\boldsymbol{V} = \{\boldsymbol{v}\}$ of the original algorithm.

Example 3.3. *Consider φ defined as in Ex. 3.2. In the first iteration of Alg. 2, we obtain $\ell = 22$ and $u = 42$, which gives $S = \{22, \ldots, 42\}$. Then, the formula $\neg\psi$ restricted to $\ell \leq \langle \boldsymbol{y} \rangle \leq u$ is analyzed to give bounds 23 and 39, and we update S to give $S \setminus \{23, \ldots, 39\} = \{22, 40, 41, 42\}$. A further restriction of φ yields an unsatisfiable formula, and thus, the output is $\alpha_{\mathsf{val}}(\varphi, \boldsymbol{y}) = \{22, 40, 41, 42\}$.*

3.1.3 Deriving Pre- and Postconditions

From now on, given an encoding $[\![b]\!]$ of a block b, we denote by $\boldsymbol{V}_{\mathsf{in}} \subseteq \boldsymbol{V}$ the inputs and by $\boldsymbol{V}_{\mathsf{out}} \subseteq \boldsymbol{V}$ the outputs of b after SSA conversion and bit-blasting. SSA conversion ensures $\boldsymbol{V}_{\mathsf{in}} \cap \boldsymbol{V}_{\mathsf{out}} = \emptyset$, which avoids accidental coupling. Given a precondition (an input state) $\mathtt{pre}_b$ that describes values of $\boldsymbol{V}_{\mathsf{in}}$, the formula

$[\![b]\!] \wedge \mathtt{pre}_b$ describes all outputs reachable from $\mathtt{pre}_b$.[1] The key idea is to compute an over-approximation of values taken by bit-vectors $\boldsymbol{V}_{\mathsf{out}}$ using Alg. 2. We define:

Definition 3.6. *Let $\boldsymbol{V} = \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\}$ and $\boldsymbol{V}_{\mathsf{in}}, \boldsymbol{V}_{\mathsf{out}} \subseteq \boldsymbol{V}$ such that $\boldsymbol{V}_{\mathsf{in}} \cap \boldsymbol{V}_{\mathsf{out}} = \emptyset$. Let B denote a finite set of basic blocks. With $b \in B$, we define maps $\mathtt{pre}_b : \boldsymbol{V}_{\mathsf{in}} \to \mathit{Val}$ and $\mathtt{post}_b : \boldsymbol{V}_{\mathsf{out}} \to \mathit{Val}$.*

It is easy to verify that the maps $\mathtt{pre}_b$ and $\mathtt{post}_b$ form a complete lattice with an appropriate (point-wise) lifting of the partial order $\sqsubseteq_{\mathsf{val}} \subseteq \mathsf{Val} \times \mathsf{Val}$.

Proposition 3.3. *Let $F = \{f : \boldsymbol{V} \to \mathit{Val} \mid f \text{ is monotone}\}$, and let $f_1, f_2 \in F$. We define $\sqsubseteq_F \subseteq F \times F$ as $f_1 \sqsubseteq_F f_2 \Leftrightarrow \forall \boldsymbol{v} \in \boldsymbol{V} : f_1(\boldsymbol{v}) \sqsubseteq_{\mathit{val}} f_2(\boldsymbol{v})$. Then, $(F, \sqsubseteq_F)$ forms a complete lattice.*

The partial order $\sqsubseteq_F$ introduced in Prop. 3.3 canonically induces maps to combine two functions $f_1, f_2 \in F$ in a lattice-theoretic setting.

Corollary 3.2. *Let $(F, \sqsubseteq_F)$ be defined as in Prop. 3.3. Then, $\sqsubseteq_F$ induces operators $\sqcup_F : F \times F \to F$ for join and $\sqcap_F : F \times F \to F$ for meet as follows:*

$$\begin{aligned} f_1 \sqcup_F f_2 &= f : \boldsymbol{V} \to \mathit{Val} \text{ such that } \forall \boldsymbol{v} \in \boldsymbol{V} : f(\boldsymbol{v}) = f_1(\boldsymbol{v}) \sqcup_{\mathit{val}} f_2(\boldsymbol{v}) \\ f_1 \sqcap_F f_2 &= f : \boldsymbol{V} \to \mathit{Val} \text{ such that } \forall \boldsymbol{v} \in \boldsymbol{V} : f(\boldsymbol{v}) = f_1(\boldsymbol{v}) \sqcap_{\mathit{val}} f_2(\boldsymbol{v}) \end{aligned}$$

Forward Interpretation

The implementation of forward interpretation is sketched in Alg. 3, taking as input a basic block b and a map $\mathtt{pre}_b$ that describes the precondition of b. First, the postcondition $\mathtt{post}_b$ is initialized with $\bot$ by mapping each $\boldsymbol{v}' \in \boldsymbol{V}_{\mathsf{out}}$ to $\bot_{\mathsf{val}} \in \mathsf{Val}$. To describe the reachable output states (the postcondition), we conjoin $[\![b]\!]$ and $\mathtt{pre}_b$ to give $\xi = [\![b]\!] \wedge \mathtt{pre}_b$. Then, for each $\boldsymbol{v}' \in \boldsymbol{V}_{\mathsf{out}}$, the algorithm computes a value set abstraction using Alg. 2, which is stored in the postcondition $\mathtt{post}_b$. The output of the procedure is a map that provides value sets for each $\boldsymbol{v}' \in \boldsymbol{V}_{\mathsf{out}}$. Intuitively, given a block $b = (b_1, \ldots, b_m)$, Alg. 3 computes the function

$$(\alpha_{\mathsf{val}} \circ \underbrace{(b_m \circ \ldots \circ b_1)}_{[\![b]\!]} \circ \gamma_{\mathsf{val}}) \quad : \quad (B \times \underbrace{(\boldsymbol{V}_{\mathsf{in}} \to \mathsf{Val})}_{\mathtt{pre}_b}) \to \underbrace{(\boldsymbol{V}_{\mathsf{out}} \to \mathsf{Val})}_{\mathtt{post}_b}$$

where $\gamma_{\mathsf{val}} : (\boldsymbol{V}_{\mathsf{in}} \to \mathsf{Val}) \to \mathbb{Z}^{|\boldsymbol{V}_{\mathsf{in}}|}$ denotes the concretization of a precondition of type $\boldsymbol{V}_{\mathsf{in}} \to \mathsf{Val}$ (which is encoded symbolically in line 2). Indeed, this formulation

[1] The alert reader will observe an inaccuracy in the formalization of Boolean formulae since we define an encoding for a block as a formula $\varphi \in \wp(\wp(\boldsymbol{V}))$, yet often assume $\boldsymbol{V} = \boldsymbol{V}_{\mathsf{in}} \cup \boldsymbol{V}_{\mathsf{out}}$. However, this inaccuracy is inconsequential. All techniques discussed in this and the following chapters, except for Alg. 2, which explicitly performs projection, provide identical results for formulae that are expressed using intermediate variables. Indeed, they are independent of intermediate variables for the same reason why projection using model enumeration is correct.

Algorithm 3 $\texttt{forward} : (B \times (\boldsymbol{V}_{\mathsf{in}} \to \mathsf{Val})) \to (\boldsymbol{V}_{\mathsf{out}} \to \mathsf{Val})$

1: $\texttt{post}_b \leftarrow \lambda \boldsymbol{v}' \in \boldsymbol{V}_{\mathsf{out}}.\bot_{\mathsf{val}}$
2: $\xi \leftarrow [\![b]\!] \wedge \left(\bigwedge_{\boldsymbol{v} \in \boldsymbol{V}_{\mathsf{in}}} \left(\bigvee_{s \in \texttt{pre}_b(\boldsymbol{v})} \langle \boldsymbol{v} \rangle = s \right) \right)$
3: **for all** $\boldsymbol{v}' \in \boldsymbol{V}_{\mathsf{out}}$ **do**
4: $\quad \texttt{post}_b \leftarrow f : \boldsymbol{V}_{\mathsf{out}} \to \mathsf{Val}$ such that $f(\boldsymbol{v}) = \begin{cases} \texttt{post}_b(\boldsymbol{v}) & : \text{if } \boldsymbol{v} \neq \boldsymbol{v}' \\ \alpha_{\mathsf{val}}(\xi, \boldsymbol{v}') & : \text{otherwise} \end{cases}$
5: **end for**
6: **return** $\texttt{post}_b$

of an abstract transformer for b dovetails with the classical design proposed by Cousot and Cousot [77, 78] and directly provides an argument for correctness.

Example 3.4. *Consider $[\![b]\!]$ defined in Ex. 3.1. Suppose the precondition $\texttt{pre}_b$ of b is defined as:*

$$\texttt{pre}_b \;=\; \big(\; \langle \boldsymbol{x} \rangle \in \{8, \ldots, 12, 15, \ldots, 18\} \;\wedge\; \langle \boldsymbol{y} \rangle \in \{0, \ldots, 22\}) \;\big)$$

The algorithm then computes:

$$\texttt{post}_b \;=\; \big(\; \langle \boldsymbol{x}'' \rangle \in \{0, \ldots, 16\}) \;\wedge\; \langle \boldsymbol{y}'' \rangle \in \{8, \ldots, 12, 15, 16\}) \;\big)$$

Backward Interpretation

Likewise, we define a backward interpreter, which computes a precondition $\texttt{pre}_b$ from a postcondition $\texttt{post}_b$ and a formula $[\![b]\!]$ that encodes a block $b \in B$. The backward analysis is derived directly from Alg. 3 using the following adaptations: (1) iterate over $\boldsymbol{v} \in \boldsymbol{V}_{\mathsf{in}}$ rather than $\boldsymbol{v}' \in \boldsymbol{V}_{\mathsf{out}}$, and (2) swap all occurrences of $\texttt{pre}_b$ and $\texttt{post}_b$, respectively. In the following, we refer to the forward analysis as $\texttt{forward} : (B \times (\boldsymbol{V}_{\mathsf{in}} \to \mathsf{Val})) \to (\boldsymbol{V}_{\mathsf{out}} \to \mathsf{Val})$, and to the backward analysis as $\texttt{backward} : (B \times (\boldsymbol{V}_{\mathsf{out}} \to \mathsf{Val})) \to (\boldsymbol{V}_{\mathsf{in}} \to \mathsf{Val})$.

Example 3.5. *Consider $[\![b]\!]$ defined as in Ex. 3.1. Put:*

$$\texttt{pre}_b \;=\; \big(\; \langle \boldsymbol{x} \rangle \in \{10, 20, 30\} \;\wedge\; \langle \boldsymbol{y} \rangle \in \{50, 60, 70\} \;\big)$$

Applying Alg. 3 in forward direction yields a postcondition:

$$\begin{aligned} \texttt{post}_b \;&=\; \texttt{forward}(b, \texttt{pre}_b) \\ &=\; \big(\; \langle \boldsymbol{x}'' \rangle \in \{2, 6, 12\} \;\wedge\; \langle \boldsymbol{y}'' \rangle \in \{4, 10, 14\} \;\big) \end{aligned}$$

By re-applying Alg. 3 in backward direction, we compute:

$$\begin{aligned} \texttt{pre}'_b \;&=\; \texttt{backward}(b, \texttt{post}_b) \\ &=\; \big(\; \langle \boldsymbol{x} \rangle \in \{4, 10, 14, 20, 26, \ldots, 244, 250, 254\} \;\wedge\; \langle \boldsymbol{y} \rangle \in \{50, 60, 70\} \;\big) \end{aligned}$$

Observe that $\texttt{pre}_b \sqsubseteq \texttt{backward}(b, \texttt{forward}(b, \texttt{pre}_b))$.

Reprise Backward interpretation leads, in certain situations, to a coarse over-approximation of the original inputs. This is the case in the above example, and has also been observed in the literature [198, 199]. The loss in precision follows from:

1. Some instructions (such as `ANL 0x05, #0x0F`) are non-invertible.

2. A more intriguing argument follows from the domain structure itself. Backward reasoning amounts to solving the following abduction problem: Given b and c, compute a non-empty a such that $(a \sqcap b) \models c$. This problem can also be seen as that of computing weakest preconditions. When a, b, and c are elements of an abstract domain, then the largest unique a with $a \neq \bot$ and $(a \sqcap b) \models c$ is called the pseudo-complement of b relative to c. A domain in which each element has a pseudo-complement is called a Heyting domain [107, Def. 4]. Unfortunately, the value set domain is not Heyting, and neither is the combination of two Heyting domains necessarily a Heyting domain [161, Prop. 4]. To illustrate, consider $b = \{0\}$ and $c = \{-5, \ldots, 5\}$, for which two incomparable a can be found, namely $a = \{-5, \ldots, -1\}$ and $a = \{1, \ldots, 5\}$. One way out of this dilemma is to lift a non-Heyting domain to its power-set domain, which yields a Boolean domain. A Boolean domain is always Heyting, since for each abstract element b, there exists a full complement a such that $a \sqcup b = \top$ and $a \sqcap b = \bot$. However, tractability then becomes an issue (cp. [200, Sect. 1]).

We thus follow the approach proposed by Rival [198] and augment φ with encodings of pre- and postconditions, respectively, to limit the loss in precision.

3.2 Program-Level Abstraction

The previous section has discussed the technical steps in computing abstractions of a single basic block in both, forward and backward direction. Most notably, these steps are bit-blasting, existential quantifier elimination, and value set abstraction. This section, in turn, extends these techniques towards a whole-program analysis using alternating forward and backward analyses. It is well-known that backward abstract interpretations can refine forward abstractions [78], an observation that is implemented in our framework. The analysis, in essence, can be seen as a classical forward fixed-point iteration for invariant generation. However, on conditional branches, backward analysis is executed on unrolled paths to refine the invariants. This strategy can be seen as a response to efficiency problems incurred by path-sensitivity. Rather than performing a path-sensitive forward analysis, we interleave the forward analysis with k-bounded, path-sensitive backward refinements.

```
1 #define N_HANDL 6
2 const UINT8 (*const code pf[])(void) = { h1, .., h6 };
3
4 UINT8 keyPress(UINT8 keyCode) {
5         if (keyCode >= N_HANDL) return C_FAIL;
6         return (*pf[keyCode]);
7 }
```

Figure 3.3: Program discussed in Ex. 3.6

3.2.1 Overview

First, the binary is disassembled using recursive traversal disassembly until an indirect jump is discovered. Recursive traversal disassemblers only map those bytes to instructions that can actually be reached from the control flow. Upon encountering an indirect jump, the disassembler stops, and a CFG is extracted from the fragment available thus far. This CFG is incomplete in the sense that it neither contains all instructions nor all edges. From the CFG, we compute a basic block representation, so that each straight-line sequence of instructions is represented by a single vertex.

Definition 3.7. *Let $G = (B, b_0, E, \sigma)$ denote a block-wise CFG, where $B = \{b_0, \ldots, b_n\}$ is the set of basic blocks, $b_0 \in B$ is the initial block, $E \subseteq B \times B$ is a transition relation, and $\sigma : E \to \wp(\wp(\boldsymbol{V}))$ labels edges with guards.*

Suppose a block $b \in B$ ends in a conditional branching instruction whose target depends on the zero flag. For both successors b_1 and b_2 of b, we have $(b, b_1) \in E$ and $(b, b_2) \in E$. Further, σ restricts (b, b_1) and (b, b_2) so that either the zero flag holds or does not hold. The analysis then outputs value sets for all input and output registers on entry and exit of each block. For the Intel MCS-51, these value sets include, most notably, the two 8-bit data pointer registers `DPL` and `DPH`, which are combined to form a 16-bit register `DP`. Together with an additive offset stored in the accumulator `A`, register `DP` indicates the target of an indirect jump. The value sets of `DPL`, `DPH`, and `A` prior to the indirect jump are thus used to guide the disassembler in the next iteration. Iterative disassembly and fixed-point computation is stopped once the jump targets stabilize. This formulation of the problem leads to a simultaneous computation of fixed points for data and control flow (cp. [142, Sect. 2.3]).

Example 3.6 (Running example)**.** *To illustrate, consider the assembly code listing in Fig. 3.4, which has been obtained from compiling the C program in Fig. 3.3. Table 3.1 presents the corresponding jump table after compilation. This program*

```
0x003: MOV 0x08, R7
0x005: MOV A, 0x08
0x007: CLR A
0x008: SUBB A, #N_HANDL
0x00A: JC C:0x00F
0x00C: MOV R7, #C_FAIL
0x00E: RET
0x00F: MOV R7, 0x08
0x011: MOV A, R7
0x012: MOV B, #0x03
0x014: MUL AB
0x015: ADD A, #0x26
0x017: MOV DPL, A
0x018: CLR A
0x019: ADDC A, #0x00
0x01B: MOV DPH, A
0x01C: AJMP C:0x038
...
0x038: CLR A
0x039: MOVC A, @(A+DP)
0x03A: MOV R3, A
0x03B: MOV A, #0x01
0x03D: MOVC A, @(A+DP)
0x03E: MOV R2, A
0x03F: MOV A, #0x02
0x040: MOVC A, @(A+DP)
0x041: MOV R1, A
0x042: AJMP C:0x100
...
0x100: MOV DPH, R2
0x102: MOV DPL, R1
0x104: CLR A
0x105: IJMP @(A+DP)
```

Figure 3.4: Assembly listing of program introduced in Ex. 3.6

illustrates a typical use of indirect control flow in embedded software. An array of function pointers `h1`, ..., `h6` *is stored in a table in program memory. These functions are indexed using an integer* `keyCode`*, which is passed to a function* `keyPress()`*.*

Block $b_{\texttt{0x003}}$ implements the comparison `keyCode >= N_HANDL` using a subtraction `SUBB A #N_HANDL`. The comparison holds true if `SUBB` does not underflow, which is indicated by the carry flag. For the success-branch, `SUBB` clears the carry and control is redirected to $b_{\texttt{0x00C}}$. The constant `#C_FAIL` is then stored in register `R7` as the return value of `keyPress()`. Otherwise, control is passed to block $b_{\texttt{0x00F}}$. This block first calculates an offset for indexing the jump table that represents `pf`. The subsequent blocks read the respective entry from the table, assign them to the data pointer `DP`, prepare the accumulator `A`, and invoke the indirect jump `IJMP @(A+DP)`.

Example 3.7 (Forward analysis without refinement)**.** *Consider again the program introduced in Ex. 3.6 and assume the abstract interpreter enters block* $b_{\texttt{0x003}}$ *with the following precondition (where bit-vector* $\boldsymbol{r_7}$ *represents register* `R7` *on entry):*

$$pre_{b_{\texttt{0x003}}} \quad = \quad \big(\ \langle \boldsymbol{r_7} \rangle \in \{1, 2, 3, 101, 102, 103\} \ \big)$$

Table 3.1: Jump table compiled for program in Fig. 3.3

Memory Address	Mark	High	Low	Function
0x26	0xFF	0x00	0x64	h1
0x29	0xFF	0x00	0x69	h2
0x2C	0xFF	0x00	0x6E	h3
0x2F	0xFF	0x00	0x73	h4
0x32	0xFF	0x00	0x78	h5
0x35	0xFF	0x00	0x7D	h6

Applying the forward abstraction procedure yields a postcondition as follows:

$$\mathit{forward}([\![b_{0x003}]\!], \mathit{pre}_{b_{0x003}}) \quad = \quad \begin{pmatrix} \langle \boldsymbol{r}'_A \rangle \in \{95, 96, 97, 251, 252, 253\} \\ \langle \boldsymbol{r}'_{0x08} \rangle \in \{1, 2, 3, 101, 102, 103\} \\ \langle \boldsymbol{c}' \rangle \in \{0, 1\} \end{pmatrix}$$

Here, $\boldsymbol{c}'$ denotes the carry flag on output of the block. Observe that the value set domain has lost the relation between values that yield $\langle \boldsymbol{c}' \rangle = 0$ and those for which we obtain $\langle \boldsymbol{c}' \rangle = 1$. Hence, all values are propagated into the successors b_{0x00C} and b_{0x00F}. Proceeding with the iteration, we eventually compute a value set $\{41, 44, 47, 85, 88, 91\}$ *for the data pointer* DPL*, which is used to index the jump table. This value set leads to the entries of functions* h2*,* h3*, and* h4 *(cp. Tab. 3.1), but also three spurious jump targets that stem from values* 85*,* 88*, and* 91 *of* DPL*.*

The overall execution of the analysis is highlighted in Fig. 3.5. The spurious values computed for register R8 on entry of block 0x00F lead to spurious values 85, 88, and 91 of DPL on exit from block 0x00F, which are computed using multiplication and addition with constants. Then, DPL and DPH are used to index the jump table. Locations 85, 88, and 91 in program memory, which do not belong to the table, hold unpredictable values, which is indicated by ? symbols in the value sets of R1, R2, and R3 on exit from block 0x038. The analysis thus infers three legitimate jump targets that indicate functions h2, h3, and h4, but also spurious ones.

It is easy to verify that the lack of precision is induced by the inability of the forward analyzer to properly capture the constraint on register R8 that is imposed by the conditional branching instruction JC C:0x00F. To sidestep this problem, we now introduce an analysis strategy which, upon branching conditions, performs a depth-bounded path-sensitive backward analysis to refine the forward invariants.

3.2.2 Forward Analysis with Invariant Refinement

Refinement-based analysis performs fixed-point iteration using a worklist in forward direction (see Alg. 4). Starting with the initial block $b_0 \in B$, blocks are analyzed

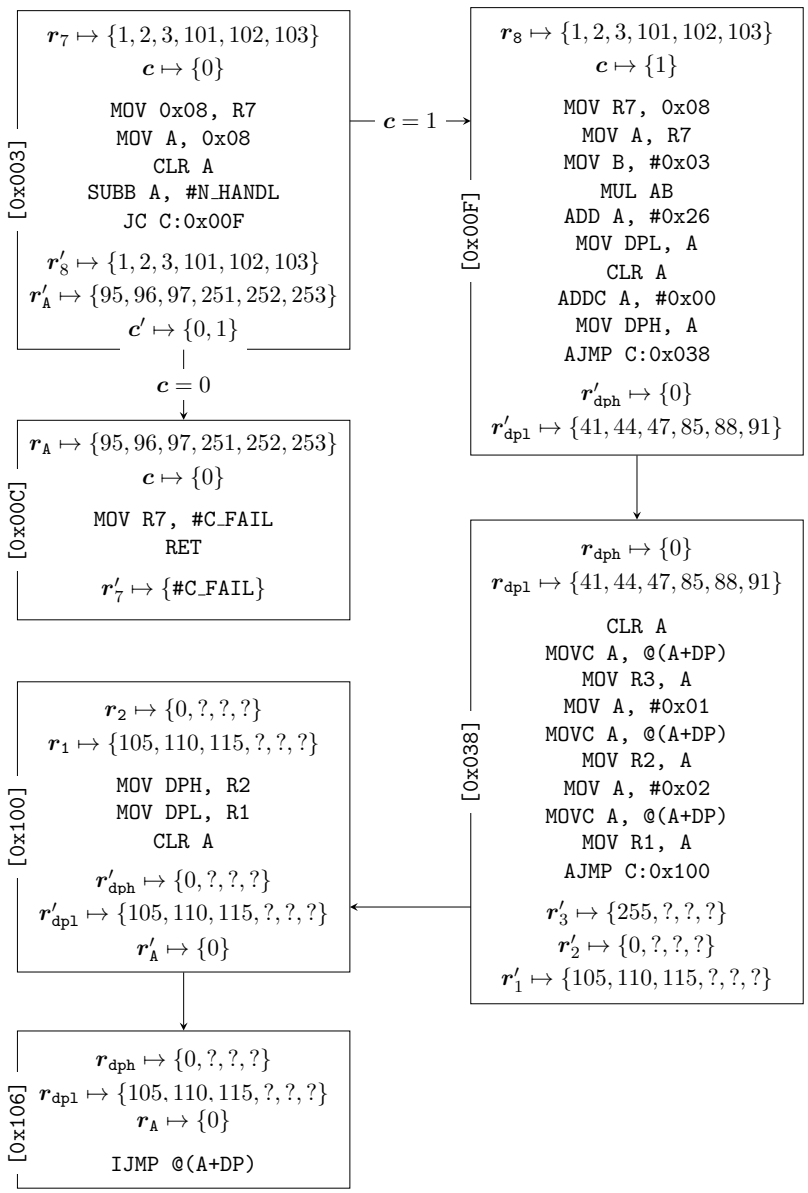


Figure 3.5: Forward value set analysis for control flow recovery; the indirect read in block `0x038` leads to three spurious values for `R1`, `R2`, and `R3`, respectively, which entail that spurious jump targets are computed in block `0x106`

Algorithm 4 `analyze`

Input: block-wise CFG (B, b_0, E, σ)
Output: pre- and postconditions $\texttt{pre}_b$ and $\texttt{post}_b$ for all $b \in B$

```
1:  W ← {b_0}
2:  while W ≠ ∅ do
3:    b ← W.pop()
4:    ξ ← forward(b)
5:    if ξ ⋢ post_b then
6:      post_b ← post_b ⊔ ξ
7:      for all s ∈ succ(b) do
8:        if σ(b, s) = true then
9:          μ ← rename(post_b)
10:       else
11:         μ ← rename(refine(s, b, k))
12:       end if
13:       if μ ⋢ pre_s then
14:         pre_s ← pre_s ⊔ μ
15:         W.add(s)
16:       end if
17:     end for
18:   end if
19: end while
```

one after another, and the outputs of each analyzed block are propagated into its successors. If the precondition $\texttt{pre}_b$ of a block $b \in B$ changes, b is added to a worklist, which indicates that b needs to be analyzed again. The key difference of the algorithm compared to standard approaches is implemented in lines 8–12. If the guard map σ does not constrain the edge $(b, s) \in E$, then the postcondition of b is propagated into the precondition of s. Here, an auxiliary procedure `rename` takes care of renaming, which is necessary to match variable names (to distinguish inputs from outputs). However, if σ imposes a constraint on (b, s), backward refinement is performed (see line 11), the result of which is propagated into $\texttt{pre}_s$. The refinement procedure is given in Alg. 5, and is discussed in what follows.

k-bounded Backward Refinement Procedure $\texttt{refine}(s, b, k)$ unrolls the block-wise CFG (B, E, b_0, σ) by k steps in backward direction, yielding a tree U in line 1. To avoid interference of `refine` with the forward analysis, we introduce auxiliary data structures $\texttt{pre}'_u$ and $\texttt{post}'_u$ to store the intermediate pre- and postconditions for each block in $u \in U$. These auxiliary data structures are used only within $\texttt{refine}(s, b, k)$. Then, the algorithm first iterates over U in pre-order and performs

Algorithm 5 $\texttt{refine}(s, b, k)$

Input: start node $s \in B$
Input: predecessor $b \in B$ of s
Input: bound $k \in \mathbb{N}$
Output: refined precondition of s

1: $U \leftarrow \texttt{unroll}(s, b, k)$
2: $\texttt{pre}'_s \leftarrow \texttt{pre}_s$
3: **for all** $(t_s, t_e) \in U$ in pre-order **do**
4: $\quad \chi \leftarrow \texttt{rename}(\sigma(t_s, t_e))$
5: $\quad \xi \leftarrow [\![t_s]\!] \wedge \chi$
6: $\quad \texttt{post}'_{t_s} \leftarrow \texttt{pre}'_{t_e}$
7: $\quad \texttt{pre}'_{t_s} \leftarrow \texttt{backward}(t_s, \texttt{post}'_{t_s}) \sqcap \texttt{pre}_{t_s}$
8: **end for**
9: **for all** $(t_s, t_e) \in U$ in post-order **do**
10: $\quad \xi \leftarrow [\![t_s]\!] \wedge \sigma(t_s, t_e)$
11: $\quad \texttt{post}'_{t_s} \leftarrow \texttt{forward}(t_s, \texttt{pre}'_{t_s}) \sqcap \texttt{post}_{t_s}$
12: $\quad \texttt{pre}'_{t_e} \leftarrow \texttt{rename}(\texttt{post}'_{t_s})$
13: **end for**
14: **return** $\texttt{pre}'_s$

a backward analysis, starting with b subject to the guard $\sigma(b, s)$ (see lines 3–8). Upon termination of the first loop, the tree is traversed in post-order by applying the forward interpreter until the starting block s is reached. Then, $\texttt{pre}'_s$ represents a more descriptive precondition of s than $\texttt{pre}_s$, i.e., $\texttt{pre}'_s \sqsubseteq \texttt{pre}_s$. This refined precondition is the output of the procedure.

Observe that the refinement procedure is parameterized by $k \in \mathbb{N}$, which provides potential for demand-driven adjustment of the refinement depth. The refinement depth too gives a way to control the trade-off between precision and tractability: Increasing k improves precision of the analysis, though at the cost of higher runtimes.

Example 3.8 (Forward analysis with refinement)**.** *For this example, let $k = 1$. The conditional branch `JC C:0x00F` passes control to either `0x00C` or `0x00F`, depending on the carry flag. Assume the precondition $\texttt{pre}_{b_{0x003}}$ as in Ex. 3.7. Of course, forward analysis yields the same postcondition $\texttt{post}_{b_{0x003}}$, too. Proceeding with block b_{0x00C}, we observe that the guard predicate $\sigma((b_{0x003}, b_{0x00C}))$ constrains the carry flag, thereby triggering refinement by calling $\texttt{refine}(b_{0x00C}, b_{0x003}, 1)$ in line 11 of Alg. 4. Then, U consists of one edge (b_{0x003}, b_{0x00C}), and backward analysis in line 7 of Alg. 5 yields:*

$$\texttt{pre}'_{b_{0x003}} \quad = \quad \big(r_7 \in \{1, 2, 3\}\big)$$

In lines 9–13*, the forward interpreter computes a refined postcondition of* b_{0x003} *as:*

$$\mathit{post}'_{b_{0x003}} \quad = \quad (\langle r'_A \rangle \in \{251, 252, 253\}, \langle r'_B \rangle \in \{1, 2, 3\}, \langle c' \rangle \in \{0\})$$

This postcondition is returned as the output of $\mathtt{refine}(b_{0x00C}, b_{0x003}, 1)$*.*

The overall results of the analysis are depicted in Fig. 3.6. After refinement starting from block `0x00F` has finished, the analysis proceeds as before and only performs forward propagation. Since `R8` now represents values 1, 2, and 3, indices 41 (`0x29`), 44 (`0x2C`), and 47 (`0x2F`) into the jump table are computed as desired (cp. Tab. 3.1). Hence, the analyzer eventually computes the entry addresses of `h2`, `h3`, and `h4`.

3.3 Experiments

We have implemented the analysis discussed in this chapter in [MC]SQUARE [207, 208] using MINISAT [94]. All experiments were performed on a desktop computer equipped with an Intel Core i5 CPU and 4 GB of RAM. To evaluate the precision of our approach, we have applied it to two sets of benchmarks for the Intel MCS-51. These benchmarks were chosen with the following questions in mind: (1) Are the runtime requirements acceptable on non-trivial examples? (2) How precisely are jump targets recovered? (3) To what extent do compiler-specific implementation details in the binary code affect the precision and performance of our framework? To evaluate the independence of our framework from a specific compiler version, we compiled all programs with both, the KEIL μVISION 3 and the SDCC 3.0 compilers.

3.3.1 Benchmarks

The first benchmark set consists of programs that explicitly use function pointers to implement typical functionalities of embedded software programs (e.g., handling of inputs from a keypad). By way of contrast, the second set implements a form of state machine which is compiled into programs that use indirect control. The sizes of the binary programs range from 52 to 180 instructions overall.

Function Pointers in Embedded C Programs

The following embedded C programs implement samples from the tutorial *Array of Pointers to Functions* in [132]. These programs use function pointers, jump tables, and pointer arithmetic.

Single Row Input The application reads input data from a bi-directional port that is connected to several buttons. Each button is associated with a handler function. The entry addresses of handler functions are stored in an array of function pointers that is indexed using an identifier of the respective button.

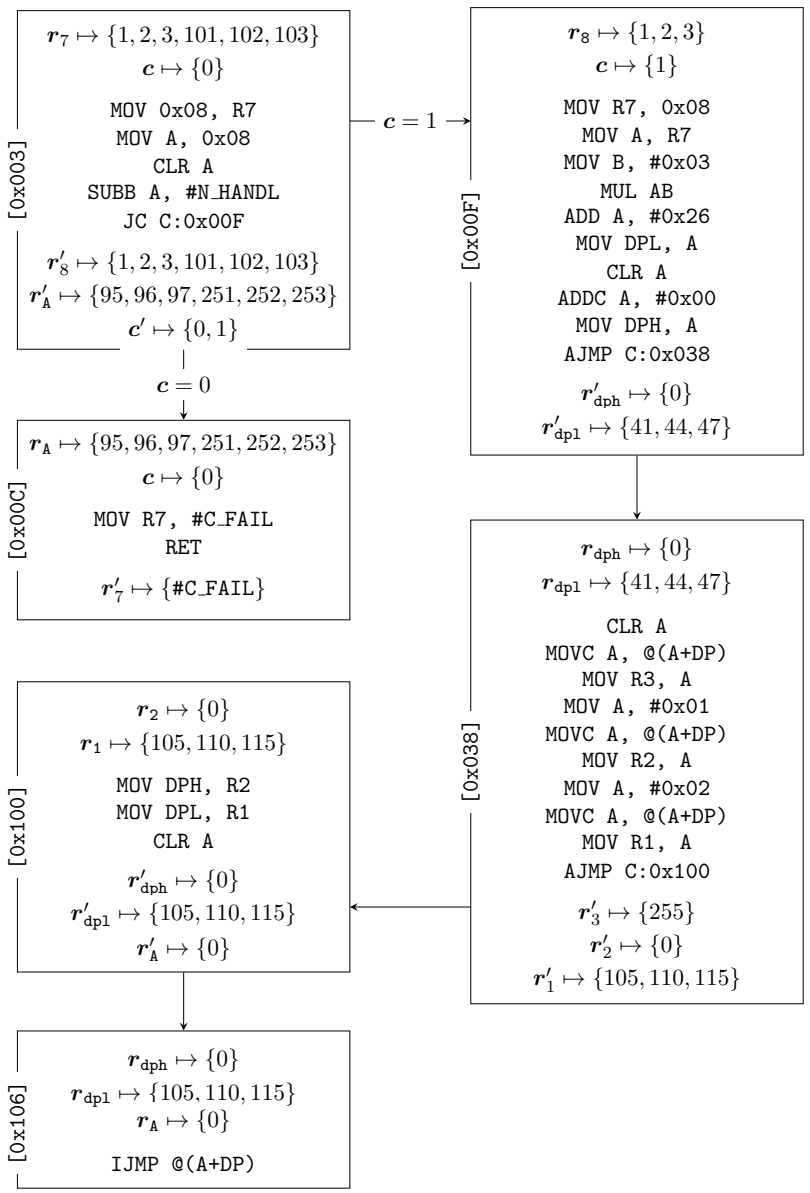


Figure 3.6: Forward value set analysis with backward refinement, which takes place in block `0x038`; with refinement, the indirect read in block `0x038` leads to exactly three jump targets in block `0x106` as desired

Keypad The application interfaces a 3×3 keypad. Whenever a key is pressed, the column and row numbers of the respective key are used to access a two-dimensional array of function pointers.

Communication Link This application handles requests that are transmitted over a serial link. It does so by indexing a table that contains callback functions to handle each command. A table-lookup paired with pointer arithmetic then determines the address of the callback function to handle the request.

Task Scheduling The application implements a low-level task scheduler. It operates on a data structure that consists of an activation interval and a function pointer to the respective task. An array holds one such entry for each task, 5 in our application. Upon each time tick, the application iterates over the array and checks whether the activation interval matches the elapsed time. The program then uses the indexed function pointer to call the respective task indirectly.

State Machines

The second set of benchmarks implements different variants of state machines using switch-case statements with the following functionality:

Single Switch-Case This program is controlled by a switch-case statement with 18 distinct cases and a default branch. A compact range of values to be tested causes the compiler to implement the switch-case statement using a jump table. The structure of this benchmark is similar to the example in Fig. 3.1, even though it is harder to analyze due to the more complex jump table.

Emergency Stop This application implements emergency stop functionality [180, pp. 40-45], which has been specified by the PLCopen consortium. The standard defines safety-related functions within the IEC 61131-3 development environment to support developers of programmable logic controllers (PLCs). The program, in essence, monitors an emergency stop button.

3.3.2 Results

Table 3.2 shows the experimental results for these benchmarks. The table clearly shows that pure forward value set analysis is insufficient for recovering jump targets precisely. Most data pointer values in forward analysis point to locations in program memory that are out of bounds, or that do not represent meaningful instructions, i.e., junk code. Yet, integrating backward refinement with a small bound (k is set to 2 on all benchmarks) eliminates the redundant jump targets for all except one benchmark (Switch Case compiled using SDCC). For Switch Case, the imprecision stems from the translation applied by SDCC, which computes the target in a loop

in which the carry flag changes so that it contains $\{0, 1\}$. The carry flag, in turn, is used to compute the jump target. Since value set analysis is non-relational, it fails to capture the relation between a concrete integer and the carry flag.[2] Jump targets are thus computed for both possible values of the carry flag, thereby leading to twice the number of jump targets. It is also interesting that in some situations combined forward and backward analysis is significantly cheaper than pure forward analysis. This is because the value sets tend to be much smaller; fewer iterations are thus required to converge onto a fixed point (since we do not use widening).

3.3.3 Comparison

Our benchmarks have also been used in a different project [20]. Bardin [16] has reported on experiments for the PPC platform using k-set analysis, the results of which are given in Tab. 3.3.[3] Here, columns *Time (PaR)* and *Time (NPaR)* denote the runtimes obtained for two different configurations of their analyzer. Their approach fails to analyze Task Scheduler and is significantly slower for Emergency Stop (135s), but is able to recover jump targets for the other benchmarks more quickly than our technique. However, the technique used by Bardin et al. [20] is driven by indirect control: it is only precise for data-flow facts relevant to control flow reconstruction, whereas our method computes precise value sets for all registers; it then uses the particular results for the `DPL`, `DPH`, and `A` registers to reconstruct a CFG, which may justify the slightly weaker performance on the other benchmarks.

3.4 Related Work

Even though the problem of control flow reconstruction has recently received increasing attention from the abstract interpretation community, it has long been studied in the context of decompilation and binary-to-source translation [58, 237]. This section surveys related approaches to decompilation, control flow reconstruction, and path-sensitive abstract interpretation, which have emerged over the past two decades.

3.4.1 Platform-Specific Decompilation

The early work of Cifuentes and Van Emmerik [61] tackles the problem of recovering jump tables from binaries. There, the authors perform slicing of the binary [60],

[2] A workaround to this problem is to augment bit-vectors that represent registers with certain bits of the status register. Value set analysis then targets bit-vectors $\boldsymbol{v} = (\boldsymbol{v}[0], \ldots, \boldsymbol{v}[w-1], \boldsymbol{c})$ to capture the relation between $\boldsymbol{v}$ and the carry flag $\boldsymbol{c}$. For further details, see [41].

[3] We have not been provided with results for the program Communication Link. Further, observe that the numbers of instructions and jump targets vary due to different compilers.

Binary Program					**Forward analysis**			**Forward + refinement**				
Name	Compiler	loc_C	instr_B	JT	RT	ST	time	RS	k	RT	ST	time
Single Row Input	Keil	80	67	6	2401	2395	2.6	2	2	6	–	3.32
	Sdcc		52		460	454	2.4	2	2	6	–	2.0
Keypad	Keil	113	113	9	3844	3835	3.49	4	2	9	–	4.33
	Sdcc		80		1508	1499	3.08	4	2	9	–	2.57
Communication Link	Keil	111	164	8	6889	6881	4.56	2	2	8	–	4.37
	Sdcc		118		84	76	3.38	2	2	8	–	4.29
Task Scheduler	Keil	81	105	5	¿1000	¿995	¿5m	17	2	5	–	14.03
	Sdcc		97					23	2	5	–	10.23
Switch Case	Keil	82	166	19	¿5000	¿4981	¿5m	94	2	19	–	17.49
	Sdcc		180		3304	3285	2.31	6	2	38	19	2.6
Emergency Stop	Keil	138	150	9	768	759	2.8	2	2	9	–	2.6
	Sdcc		141		256	247	2.9	2	2	9	–	3.1

Table 3.2: Experimental results for pure forward analysis as well as combined forward and backward analysis. The 1st category *Binary program* presents statistics about the analyzed program: column loc_C gives the lines of code in C, whereas $Instr_B$ indicates the number of instructions in the binary program; column JT contains the overall number of dynamic jump targets. The 2nd category contains results for pure forward analysis, where RT displays the number of recovered targets, followed by the number of spurious targets ST; column *Time* gives the overall runtime, with a timeout of 5 minutes. The 3rd category *Forward + refinement* additionally contains the number of refinement steps RS and the depth-bound k.

Table 3.3: Experiments reported by Bardin [16]

Name	instr_B	JT	RT	ST	Time (PaR)	Time (NPaR)
Single Row Input	158	6	6	6	$< 1s$	$< 1s$
Keypad	224	8	8	8	$< 1s$	$< 1s$
Task Scheduler	127	3	0	∞	$2s$	$< 1s$
Switch Case	204	19	19	19	$< 1s$	$< 1s$
Emergency Stop	475	10	10	10	$135s$	$17s$

where the slicing criterion is determined by the indirect jump instruction. The remaining program slice is then transformed into a normal form using rewriting techniques, thereby explicitly tackling switch-case statements, which often exhibit a similar structure in the binary. Srivastava and Wall [227] introduced the concept of *hell nodes* to control flow analysis. If the target of an indirect jump is unknown, then control is redirected to an auxiliary hell node, which is always reachable and for which all abstract program properties are initialized to $\top$. This concept was resumed by De Sutter et al. [91], whose analysis starts with a conservative CFG in which every indirect control instruction targets the hell node. The conservative approximation of the CFG is then incrementally refined using static analyses such as constant propagation, the information of which is used to replace some control flow edges to hell nodes by regular edges. This approach was implemented in the Alto link-time optimizer for Alpha processors [174]. By way of contrast, Theiling [232] uses a bottom-up analysis, starting with a collection of entry points, based on which a CFG is incrementally extended. His approach, however, requires the compiler to be known [232, Sect. 5.2] to find switch tables, which is not required for our algorithm as we *compute* the addresses that represent jump tables. Holsti [129] uses partial evaluation to analyze switch tables, but also relies on compiler-specific information to find the respective handlers [129, Sect. 7]. The de-facto industrial standard for binary analysis is IDA Pro [128]. This tool is based on unsound heuristics, such as assuming that every call returns to its original call-site, which can lead to erroneous edges in the CFG. Furthermore, the generated CFG is incomplete in general, based on limited property propagation mechanisms (constants, e.g., are only propagated within basic blocks).

3.4.2 Control Flow Reconstruction by Abstract Interpretation

The most prominent tool for control flow reconstruction by abstract interpretation is probably Jakstab [141], which interleaves disassembly with abstract interpretation as we do. The abstract domain used in Jakstab consists of a form of symbolic constant lattice [142, Sect. 4]. Later, Flexeder et al. [100] extended their algorithm

towards interprocedural analysis. Recently, Kinder and Kravchenko [140] proposed to alternate between over-approximation using abstract interpretation and under-approximation using simulation. Under-approximation yields stricter preconditions on indirect control instructions, from which abstract interpretation benefits. By way of contrast, Bardin et al. [20] use the k-set abstract domain paired with refinement. The authors refer to this approach as *value analysis with precision requirements*. The key idea of their work is to refine the precision of the abstract domain once the chosen k-set becomes imprecise (in that it yields $\top$). CodeSurfer/x86 [9, 10] uses IDA Pro to get access to executable files and then employs heuristic analyses using abstract domains such as affine equalities and strided intervals. As in IDA Pro, the resulting CFG may be incomplete [9, Sect. 3.8].

3.4.3 Control Flow Reconstruction in Model Checking and Testing

McVeto [231] performs reachability analysis of binaries using directed proof generation [121]. In McVeto, the binary is decompiled instruction by instruction and the state space is explored using on-the-fly predicate refinement, starting with an initial abstraction of the binary that consists of two propositions about the program counter. This abstraction is motivated by the fact that, in contrast to source code model checking, the CFG of the program is not initially available, and thus has to be extended gradually. Our own tool [mc]square [207, 208, 211] uses explicit-state model checking based on simulation of the hardware, and can thus directly extract jump targets from the state space. Control flow analysis also appears in automated (concolic) testing of executables. The tool Osmose [17, 18] derives test-cases based on an incomplete CFG, which is then incrementally extended using jump targets that occur in executions of a test. The CFG may thus still be incomplete, but converges onto a sound one from below while test-cases are explored. Our own work on systematic test-case generation for binaries [190] uses the same approach.

3.4.4 Path-Sensitive Abstract Interpretation

In many practical cases, non-relational and convex abstract domains are not expressive enough to capture required invariants. There has thus been much interest in refining abstract domains to improve the accuracy of abstract interpretations. Cousot and Cousot [78] have introduced disjunctive completion, which can be seen as a form of lifting an abstract domain to its power-set representation. The drawback of this construction, however, is its computational cost as the size of the domain increases exponentially. Rival and Mauborgne [200] introduced trace partitioning to augment the abstract analysis domain with a finite partitioning of the possibly infinite set of paths. The partitioning is determined heuristically or delegated to the end-user. Different variations of this approach, such as control flow splitting

using Boolean flags [220], have been proposed. A different direction was followed by Balakrishnan et al. [11], who combined forward and backward analysis so as to eliminate infeasible paths, thereby obtaining a refined control structure of the program. Similar in spirit is the work of Rival [198, 199] on error localization using abstract interpretation by applying backward transformers. To limit the imprecision that stems from applying transformers in reverse, Rival intersects backward states with those obtained using forward analysis, which is not dissimilar to our approach of encoding pre- and postconditions within the SAT instance.

Later work by Balakrishnan et al. [12] uses classical abstract interpretation for control structure refinement, where paths through a loop are represented by regular expressions. Our own contribution to this field [38] extends the work of Balakrishnan et al. [12] in two ways: (a) symbolic best transformers [197] for each path through the loop are generated, which (b) explicitly model bit-vector arithmetic. The method described in this chapter differs from the above works — which apply some form of domain or control flow refinement — by using bounded path-sensitive backwards analysis to refine certain abstract elements.

3.5 Discussion

In this chapter, we have explored the possibility of using SAT solvers for value set analysis of binary code with applications in control flow reconstruction. The framework allows us to treat forward and backward analyses uniformly, based on a relational description of the program semantics. The algorithms that compute forward and backward value set abstractions, which are based on a mixture of existential quantification and incremental SAT solving, then exhibit a symmetric structure.

The key to precision lies in the combination of these analyses, which can be seen as a response to the tractability issues associated with path-sensitivity. Using k-bounded backward refinement entails that recovery of precision is performed locally, independently of the size of the entire program. This approach mitigates the computational problems incurred by bit-precise path-sensitive techniques. It is well-known since the early works of Cousot and Cousot [77, 78] that backward abstract interpretations can refine forward analyses. However, the difficulty of designing precise backward transformers together with limitations of domains frequently used in abstract interpretation — recall the discussion on structural properties of Heyting domains — has thus far often thwarted the practical application of backward refinement. We believe that automatic abstraction using uniform forward and backward analyses may provide a technique to bridge this gap between theoretical results and practical applications.

4 Automatic Abstraction of Bit-Vector Formulae

The technique for control flow reconstruction based on value set analysis presented in the previous chapter invokes a SAT solver on each application of a transfer function. Although different in its technical details, the approach can thus be seen as similar in spirit to the algorithm of Reps et al. [197, Sect. 2] to implement symbolic best transformers; Alg. 6 presents a variation of their technique.[1] The procedure $\texttt{sbt}(\varphi, G)$ takes as input a formula φ and a Galois connection $G = (C, \gamma, \alpha, D)$, which relates a concrete domain C and an abstract domain D through a concretization map $\gamma : D \to C$ and an abstraction map $\alpha : C \to D$ (cp. [77, Sect. 6]). The output of $\texttt{sbt}(\varphi, G)$ is the most precise abstract element $d \in D$ that over-approximates all concrete states $c \in C$ described by φ.[2] The key idea of Reps et al. is to invoke a decision procedure — in their method a theorem prover — on each application of a transfer function so as to compute the most descriptive abstract output directly from the concrete semantics of a program statement. In the worst case, this approach requires h calls to a decision procedure to evaluate a transfer function, where h is the chain-length of the respective abstract domain D [197, Thm. 1]. In the previous chapter, we have shown that — owing to efficient SAT solvers — this dynamic approach is tractable if only few calls to the solver are required to compute the output: the 8-bit value set domain paired with a dedicated abstraction procedure exhibits this desirable property. However, this is not always the case.

Drawbacks of Online Transformers Indeed, many abstract domains exhibit chains whose length is polynomial or exponential in the number and domains of involved variables (variable domains are discrete in our setting, and thus finite), which may render the online computation of transformers during fixed-point iteration intractable. We support this observation by means of an example.

[1] Reps et al. [197] additionally use a map $\beta(\mathbf{m})$ to transform a concrete model $\mathbf{m}$, which is defined as map $\mathbf{m} : \mathsf{Var} \to C$, into an abstract store $\mathsf{Var} \to D$. However, this is only a minor technical difference that we omit for the purpose of presentation.

[2] Requiring a Galois connection $G = (C, \gamma, \alpha, D)$ to relate the concrete domain C and the abstract domain D guarantees the computation of a best abstraction. Intuitively, a Galois connection entails that there is an optimal abstract analogue $d \in D$ for each $c \in C$. If this requirement is dropped, the algorithm produces a sound abstraction which may be suboptimal.

Algorithm 6 $\texttt{sbt}(\varphi, G)$

Input: formula φ
Input: Galois connection $G = (C, \gamma, \alpha, D)$
Output: symbolic best abstraction d of φ in D

```
1: d ← ⊥
2: ψ ← φ
3: while ψ is satisfiable do
4:     m ← model of ψ
5:     d ← d ⊔ α(m)
6:     ψ ← ψ ∧ ¬γ(d)
7: end while
8: return d
```

Example 4.1. *Consider octagons [166], which consist of conjunctions of inequalities of the form* $\pm\boldsymbol{v}_1 \pm \boldsymbol{v}_2 \leq c$ *where* $\boldsymbol{v}_1$ *and* $\boldsymbol{v}_2$ *are bit-vectors and* $c \in \mathbb{Z}$*. For simplicity, assume that a formula* $\varphi \in \wp(\wp(\{\boldsymbol{v}_1, \boldsymbol{v}_2\}))$ *describes the following set of models:*

$$\{(0,0),(0,1),(1,1),(1,2),\ldots,(254,255),(255,255)\} \in \wp(\mathbb{Z}^2)$$

The formula could thus be abstracted by $0 \leq \langle \boldsymbol{v}_1 \rangle + \langle \boldsymbol{v}_2 \rangle \leq 500$*. Alg. 6 queries a solver to obtain a model* $\mathbf{m}_1$ *of* φ*, e.g.,* $\mathbf{m}_1 = \{\langle \boldsymbol{v}_1 \rangle = 0, \langle \boldsymbol{v}_2 \rangle = 0\}$*, which induces* $0 \leq \langle \boldsymbol{v}_1 \rangle + \langle \boldsymbol{v}_2 \rangle \leq 0$*. In a second iteration,* $\varphi \wedge \neg(0 \leq \langle \boldsymbol{v}_1 \rangle + \langle \boldsymbol{v}_2 \rangle \leq 0)$ *is passed to the solver, possibly yielding a model* $\mathbf{m}_2 = \{\langle \boldsymbol{v}_0 \rangle = 0, \langle \boldsymbol{v}_2 \rangle = 1\}$ *that defines a relaxed constraint* $0 \leq \langle \boldsymbol{v}_1 \rangle + \langle \boldsymbol{v}_2 \rangle \leq 1$*. Proceeding like before, the algorithm requires* 499 *more calls to the decision procedure to converge onto* $0 \leq \langle \boldsymbol{v}_1 \rangle + \langle \boldsymbol{v}_2 \rangle \leq 500$*.*

Observe, however, that the algorithm could have produced the abstraction $0 \leq \langle \boldsymbol{v}_1 \rangle + \langle \boldsymbol{v}_2 \rangle \leq 500$ in three iterations, had it produced models $\mathbf{m}_1 = \{\langle \boldsymbol{v}_1 \rangle = 0, \langle \boldsymbol{v}_2 \rangle = 0\}$ and $\mathbf{m}_2 = \{\langle \boldsymbol{v}_1 \rangle = 255, \langle \boldsymbol{v}_2 \rangle = 255\}$ that define $0 = \langle \boldsymbol{v}_1 \rangle + \langle \boldsymbol{v}_2 \rangle$ and $\langle \boldsymbol{v}_1 \rangle + \langle \boldsymbol{v}_2 \rangle = 500$, respectively. Afterwards, the solver is invoked once more to prove that the abstraction is sound by testing $\varphi \wedge \neg(0 \leq \langle \boldsymbol{v}_1 \rangle + \langle \boldsymbol{v}_2 \rangle \leq 500)$ for *unsatisfiability*. Computing optimal abstract outputs with a dependency on the solving strategy as strong as in Alg. 6 leads to unpredictable runtimes which even may vary strongly from one execution of the analysis to another; of course, this is highly undesirable.

Offline Computation of Abstractions As an alternative to computing symbolic best transformers online, we propose to generate transfer functions offline, prior to the analysis itself. Then, a decision procedure is repeatedly invoked to compute a structure that describes how abstract inputs are mapped to abstract outputs. Performing program analysis then merely amounts to applying the computed map, rather than invoking the solver during the analysis. However, it is important

to observe that a propositional formula φ, which can be derived to represent the concrete semantics of a program statement or a basic block, does not prescribe how to compute a transfer function. Therefore, we show how to augment φ with additional constraints so that an abstraction can be extracted from φ; this is important to efficiently abstract φ using linear constraint domains such as octagons or convex polyhedra. Most notably, the abstract domains studied in this chapter include affine equalities [136], arithmetical congruences [114], polynomials of bounded degree [73], octagons [166], and convex polyhedra [82].

Roadmap Overall, this chapter provides a collection of techniques for inferring abstractions for basic blocks that are defined as bit-vector relations. If programs are defined over finite integers, then over- and underflows manifest themselves somewhere: if not in the abstract domain [115, 144, 171, 172] then in the transformer [31, 32] or elsewhere [223]. In general, abstract domains that model unbounded integer arithmetic are attractive because of their structural simplicity, compared to those that support wrap-around arithmetic.[3] When describing bit-vector programs using abstract domains defined over unbounded integers (such as affine equalities or octagons), one problem is thus to provide a systematic way to model wrapping arithmetic within the transfer functions.[4]

To illustrate the problem, consider the domain of affine relations and the operation `ADD R0 R1`, which computes the sum of two inputs `R0` and `R1` and stores the result in `R0`. Driven by intuition, one might be tempted to model this instruction using an equality $\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle$. Yet, such modeling is unsound as it does not address the effects of modular arithmetic. Indeed, no affine equality can model `ADD R0 R1` for all feasible inputs, its abstraction is thus $\top_{\mathsf{aff}}$. With respect to wrap-around arithmetic, each instruction can have three modes of operation, depending on whether it overflows, underflows, or does neither (called regular). An instruction with more than one mode of operation is called multi-modal, as opposed to uni-modal instructions (such as `MOV R0 R1`). Which mode is applicable then depends on the values of variables on entry to the instruction. Knowing that a particular mode is applicable permits a specialized transfer function to be applied for inputs which conform to that mode. To illustrate, consider again `ADD R0 R1` on an 8-bit machine. This operation exhibits three different affine relations, depending on its mode:

$$\begin{array}{lcl} \text{regular} & : & \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \\ \text{overflow} & : & \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle - 256 \\ \text{underflow} & : & \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle + 256 \end{array}$$

[3] It is, for example, unclear how to model inequalities in conjunctive modular domains [172, Sect. 7].

[4] An alternative approach is implemented in static analyzers such as Astrée [83–85]. There, the program semantics is modeled over unbounded integers, followed by checks for over- and underflow. However, this approach is not suitable for the analysis of programs for 8-bit architectures where, e.g., 16-bit comparisons intentionally depend on over- and underflows.

Hence, when applying abstract domains such as affine equalities to describe finite bit-vector semantics, it is important to separate modes of operations. These modes can, in turn, be specified as linear (octagonal) constraints over the inputs:

$$\begin{array}{lcrcrcr} \text{regular} & : & -128 & \leq & \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle & \leq & 127 \\ \text{overflow} & : & 128 & \leq & \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle & \leq & 254 \\ \text{underflow} & : & -256 & \leq & \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle & \leq & -129 \end{array}$$

This observation leads to a formulation of transfer functions as systems of guarded updates to separate and describe the different modes of operation. Then, the guard characterizes an over-approximation of those inputs that satisfy the respective mode, and therefore, which type of update is applicable. By way of contrast, the update can be seen as an input-output transformer that stipulates how inputs are mapped to outputs in an applicable mode. However, since a block is composed of several instructions, each of which may have different modes of operation, this necessitates an automatic technique that identifies the feasible mode combinations, for each of which it computes an abstraction. The latter involves two technical challenges: given a formula φ that describes the semantics of a basic block subject to a fixed mode combination, it is necessary to (1) compute a precondition of φ (the guard) using some class of inequality, and (2) derive a direct relationship between the inputs and the outputs of the block (the update).

Outline This chapter is concerned with computing abstractions of (a subset of) the registers characterized by a block. We present techniques for abstracting the semantics of blocks as follows:

- First, Chap. 4.1 discusses the technical details of automatically separating different modes of operation in bit-vector formulae.
- Then, Chap. 4.2 presents techniques that abstract relations between variables described by a formula with affine equalities, arithmetical congruences, bounded polynomials, octagons, and convex polyhedra.
- Following, Chap. 4.3 presents a technique that we refer to as extrapolation: the key idea of this method is to derive abstractions for short bit-vectors (e.g., in a 5-bit representation) and then soundly extrapolate the derived relations towards larger bit-vectors (e.g., 32 bits) to improve efficiency.

The chapter concludes with experimental results in Chap. 4.4 and a discussion in Chap. 4.5. The content of this chapter is strongly related to the following Chap. 5, which is concerned with deriving transformers for symbolic constraints as opposed to computing abstract relations among variables. However, the techniques presented in Chap. 5 are to a great extent based on the abstraction mechanisms discussed herein. We therefore present related work for both, Chap. 4 and Chap. 5, in Chap. 5.6.

```
1 : ADD R0 R1;    2 : MOV R2 R0;    3 : EOR R2 R1;    4 : LSL R2;
5 : SBC R2 R2;    6 : ADD R0 R2;    7 : EOR R0 R2;
```

Figure 4.1: Example assembly listing for AVR microcontrollers

4.1 Separation of Modes

As in Chap. 3.1.1, we bit-blast a block $b = (b_1, \ldots, b_n)$ consisting of n instructions by composing it from n formulae $[\![b_i]\!] \in \wp(\wp(\boldsymbol{V}))$ to give $[\![b]\!] = \bigwedge_{i=1}^{n} [\![b_i]\!] \in \wp(\wp(\boldsymbol{V}))$. Each instruction in b can operate in one out of at most three modes: it overflows, underflows, or does neither. Of course, the operations $b_1, \ldots, b_n$ that constitute b may operate in different modes, although the mode of one instruction may preclude a mode of another from being applicable. A mode m_{i,k_i} is then chosen for each instruction b_i, and a single formula is constructed by augmenting $[\![b]\!]$ with encodings $[\![m_{i,k_i}]\!] \in \wp(\wp(\boldsymbol{V}))$ of constraints imposed by each mode, which yields a formula:

$$\underbrace{\left(\bigwedge_{i=1}^{n} [\![b_i]\!]\right)}_{\text{semantics of } b} \quad \wedge \quad \underbrace{\left(\bigwedge_{i=1}^{n} [\![m_{i,k_i}]\!]\right)}_{\text{mode constraints}} \quad \in \quad \wp(\wp(\boldsymbol{V}))$$

Observe that the modes of each instruction and the respective encodings can directly be derived from the instruction-set specification of the target hardware. If the composed formula is unsatisfiable, then the mode combination is inconsistent. Otherwise, the mode combination is feasible and the formula describes one type of wrapping (or non-wrapping) behavior that can be realized within the block.

Example 4.2. *On an 8-bit architecture, instruction* `ADD R0 R1` *from Fig. 4.1 is encoded as* $[\![$`ADD R0 R1`$]\!]$*, which is given propositionally as:*

$$\begin{aligned}&(\textstyle\bigwedge_{i=0}^{7} \boldsymbol{r0}'[i] \leftrightarrow (\boldsymbol{r0}[i] \oplus \boldsymbol{r1}[i] \oplus \boldsymbol{c}[i])) \wedge \\ &\neg \boldsymbol{c}[0] \wedge (\textstyle\bigwedge_{i=0}^{6} \boldsymbol{c}[i+1] \leftrightarrow ((\boldsymbol{r0}[i] \wedge \boldsymbol{r1}[i]) \vee (\boldsymbol{r0}[i] \wedge \boldsymbol{c}[i]) \vee (\boldsymbol{r1}[i] \wedge \boldsymbol{c}[i])))\end{aligned}$$

Here, the bit-vector $\boldsymbol{c}$ *denotes auxiliary carry-bits to simplify the encoding. The* `ADD` *instruction has three modes of operation: it can overflow (m_O), underflow (m_U), or behave regularly ($m_R = \neg m_O \wedge \neg m_U$). These modes are encoded as follows:*

$$\begin{aligned}[\![m_O]\!] &= \neg \boldsymbol{r0}[7] \wedge \neg \boldsymbol{r1}[7] \wedge \boldsymbol{r0}'[7] \\ [\![m_U]\!] &= \boldsymbol{r0}[7] \wedge \boldsymbol{r1}[7] \wedge \neg \boldsymbol{r0}'[7] \\ [\![m_R]\!] &= (\boldsymbol{r0}[7] \vee \boldsymbol{r1}[7] \vee \neg \boldsymbol{r0}'[7]) \wedge (\neg \boldsymbol{r0}[7] \vee \neg \boldsymbol{r1}[7] \vee \boldsymbol{r0}'[7])\end{aligned}$$

For example, $[\![$`ADD R0 R1`$]\!] \wedge [\![m_O]\!]$ *describes the semantics of* `ADD R0 R1` *for those inputs which lead to an overflow. The set of models of* $[\![$`ADD R0 R1`$]\!] \wedge [\![m_O]\!]$ *contains*

Table 4.1: Feasible and infeasible modes for the basic block in Fig. 4.1; only 5 out of 18 different mode combinations are feasible, which is checked by testing the respective encoding for satisfiability

`ADD R0 R1`	`LSL R2`	`ADD R0 R2`	feasible?
R	R	R	**yes**
R	R	O	no
R	R	U	no
R	O	R	**yes**
R	O	O	no
R	O	U	no
O	R	R	no
O	R	O	no
O	R	U	no
O	O	R	**yes**
O	O	O	no
O	O	U	**yes**
U	R	R	no
U	R	O	no
U	R	U	no
U	O	R	**yes**
U	O	O	no
U	O	U	no

exactly those valuations for which the arithmetic sum of `R0` and `R1` yields a value greater than or equal to 128 (if `R0` and `R1` are interpreted as signed bit-vectors). Likewise, $[\![$`ADD R0 R1`$]\!] \wedge [\![m_U]\!]$ *and* $[\![$`ADD R0 R1`$]\!] \wedge [\![m_R]\!]$ *describe the semantics of `ADD R0 R1` for underflow and regular operation, respectively.*

4.1.1 Detecting Feasible Modes

Consider again the assembly code listing in Fig. 4.1. The instruction `LSL R2` shifts register `R2` to the left by one bit; the most significant bit of `R2` is moved into the carry flag. If the carry flag is set on output of the instruction, an overflow occurs. However, there is no underflow specified for `LSL`, the instruction thus has two modes of operation. The instruction `SBC` (subtract with carry) is multi-modal as it can over- or underflow. Yet, in the case of two equal operands, the instruction `SBC R2 R2` can only result in `R2` $= 0$ or `R2` $= -1$, depending on the carry flag on input. In essence, `SBC R2 R2` stores the carry flag on input in every single bit of `R2`. We thus ignore the wrapping of `SBC R2 R2` and consider it to be uni-modal. `EOR` and

`MOV` are both uni-modal, too. Table 4.1 shows the feasible modes for the listing in Fig. 4.1. The second row, e.g., refers to the formula which encodes the block paired constraints for regular behavior of `ADD R0 R1` and `LSL R2` and overflow of `ADD R0 R2`. Unsatisfiability entails that this combination is infeasible. Since `ADD R0 R1` and `ADD R0 R2` both have three modes, whereas `LSL R2` has two, the above block constitutes $3 \cdot 2 \cdot 3 = 18$ mode combinations. Yet, only five of these combinations are feasible. It is thus necessary to derive abstractions only for these feasible modes.

4.1.2 Incremental Feasibility Checks

The number of mode combinations in a single block is, in the worst case, exponential in the number of instructions. The number of calls to a decision procedure required to determine feasible modes is thus exponential, too. Further, incrementality, which greatly affects the efficiency of contemporary solvers, cannot be exploited when feasibility of mode combinations is checked one by one. We therefore present an alternative strategy for checking feasibility of mode combinations incrementally.

Example 4.3. *Let φ encode the block in Fig. 4.1 as before and consider the case where `ADD R0 R1` underflows and `LSL R2` behaves regularly. The formula*

$$\varphi' \quad = \quad \varphi \wedge [\![m_{\texttt{ADD R0 R1},U}]\!] \wedge [\![m_{\texttt{LSL R2},R}]\!]$$

describes this compound mode, independently of the second `ADD`. Observe that $\varphi' \wedge [\![m_{\texttt{ADD R0 R2},O}]\!] \models \varphi'$, $\varphi' \wedge [\![m_{\texttt{ADD R0 R2},U}]\!] \models \varphi'$, and $\varphi' \wedge [\![m_{\texttt{ADD R0 R2},R}]\!] \models \varphi'$. From unsatisfiability of φ', we thus deduce that φ' equipped with a constraint on the mode of `ADD R0 R2` is infeasible, too.

The above example suggests extending the formula φ with mode constraints, such as $[\![m_{\texttt{ADD R0 R2},O}]\!]$, instruction by instruction. This strategy can also be seen as analyzing modes in a tree-like fashion. A sub-tree, which represents different modes of one instruction, is then explored iff the formula representing its root is satisfiable, as illustrated in Fig. 4.2. This technique may increase the overall number of SAT instances to be solved (21 instead of 18 for the running example), because all leaves are reachable in the worst case. However, the tree-like strategy integrates smoothly with incremental solving techniques since the additional mode constraints can be passed as assumptions, thereby permitting the solver to reuse information learnt.

Discussion Which solving strategy for feasible modes outperforms the other depends on the distribution of feasible mode combinations; this distribution, in turn, depends on the intrinsics of the analyzed block. There is thus no clear winner. Of course, one could execute both strategies in parallel with the possibility of communicating infeasible modes.

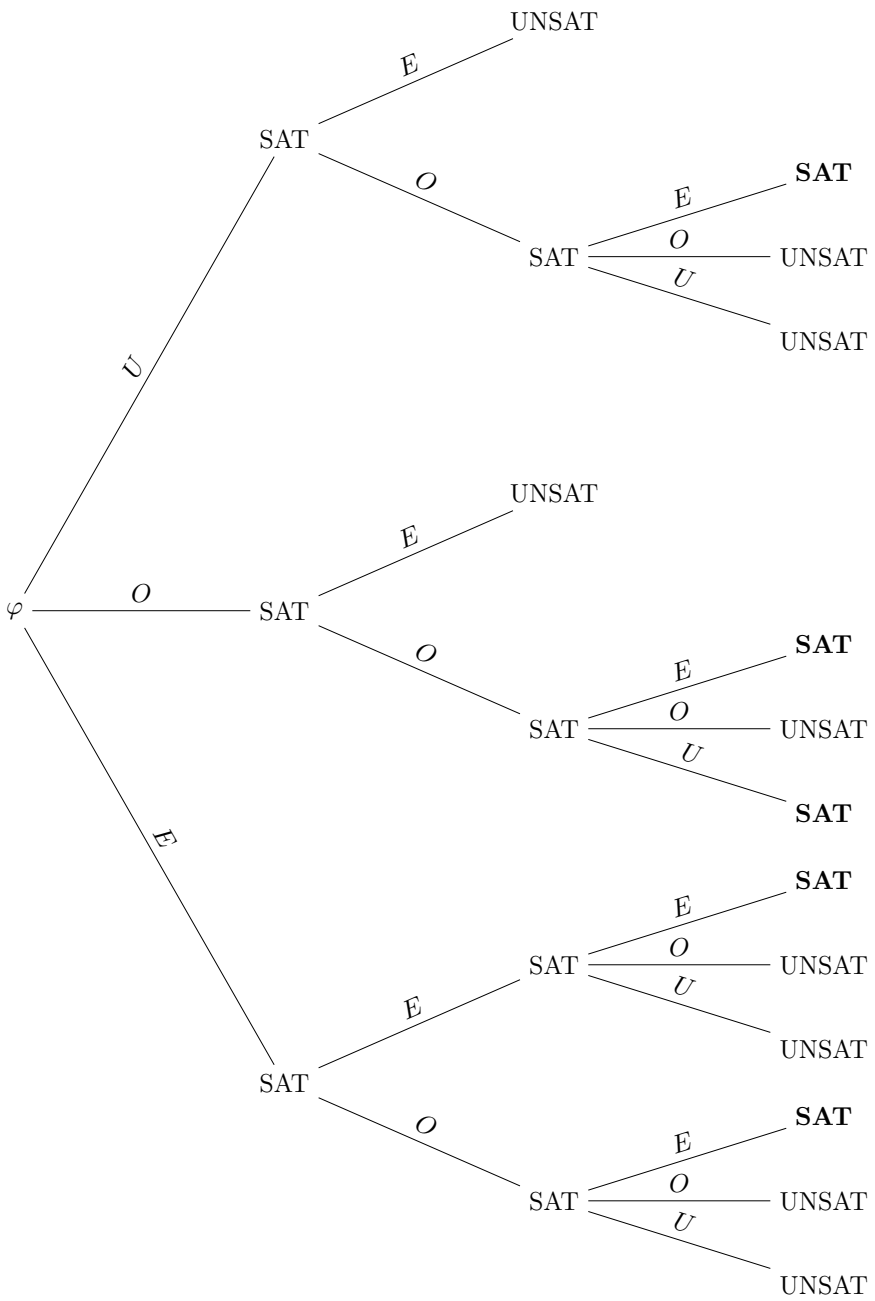


Figure 4.2: Incremental detection of feasible mode combinations; each level in the tree corresponds to one column in Tab. 4.1 (from left to right)

4.2 Symbolic Abstractions for Bit-Vectors

For a formula that encodes a feasible mode combination, we need to compute an abstraction that mimics the concrete semantics of the chosen mode combination. As argued before, we model transformers for bit-vector relations as guarded updates. This section discusses different classes of abstractions required for this representation: (1) octagons and polyhedra, which are used to describe linear inequalities among registers in the respective mode combination, and (2) affine and polynomial abstractions that represent equality relations between variables. Additionally, we discuss a technique that infers arithmetical congruences that hold for single registers, yielding information about strides of values. The techniques to compute such abstractions are discussed from Chap. 4.2.1 to Chap. 4.2.6.

4.2.1 Octagons

Octagons are formed from conjunctions of constraints $\lambda_1 \cdot \langle\!\langle \boldsymbol{v}_1 \rangle\!\rangle + \lambda_2 \cdot \langle\!\langle \boldsymbol{v}_2 \rangle\!\rangle \leq d$ where $\lambda_1, \lambda_2 \in \{-1, 1\}$, $d \in \mathbb{Z}$, and $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ are variables. In the following, we sometimes abbreviate this form of inequality as $\pm\langle\!\langle \boldsymbol{v}_1 \rangle\!\rangle \pm \langle\!\langle \boldsymbol{v}_2 \rangle\!\rangle \leq d$. Octagons can thus be seen as a sub-class of convex polyhedra with the inequalities drawn from a finite set of fixed templates. We denote the domain of such conjunctions of inequalities by Oct. Additionally, $\sqcup_{\mathsf{oct}} : \mathsf{Oct} \times \mathsf{Oct} \to \mathsf{Oct}$ denotes the join of two octagons, and the respective partial order $\sqsubseteq_{\mathsf{oct}} \subseteq \mathsf{Oct} \times \mathsf{Oct}$ is induced by entailment. Interestingly, octagons are often sufficiently expressive to verify interesting properties of programs (cp. [166, Sect. 1.1]). Further, if octagons are represented using difference bounds matrices (DBMs) to encode potential constraints [166, Sect. 2.2], then all relevant domain operations can be expressed in $\mathcal{O}(n^3)$ where n is the number of variables, as opposed to the exponential complexity incurred by convex polyhedra [166, Sect. 4.8].[5] Using this representation, the join $\sqcup_{\mathsf{oct}}$ of two octagons, for instance, can then be expressed as the point-wise maximum of the elements of their DBMs [166, Sect. 2.3]. Expressiveness of the domain and efficiency of the domain operations may thus explain the popularity of the octagon abstract domain.

Abstraction by Dichotomic Search

Suppose a formula $\varphi \in \wp(\wp(\boldsymbol{V}))$ over bit-vectors $\boldsymbol{V} = \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\}$ has k satisfying assignments, which are given by maps $\mathbf{m}_i : \boldsymbol{V} \to \mathbb{Z}$ for $1 \leq i \leq k$. Then, each $\mathbf{m}_i$ can equivalently be represented as an element $m_i = (\mathbf{m}_i(\boldsymbol{v}_1), \ldots, \mathbf{m}_i(\boldsymbol{v}_n)) \in \mathbb{Z}^n$. Intuitively, abstraction α_{oct} is then defined as the operation that yields the least

[5] In his seminal work, Miné [166, Sect. 3.5] presents a so-called tight closure operation for integral octagons which has complexity $\mathcal{O}(n^4)$. The key idea of this algorithm is to first close a real-valued octagonal system, which has cubic complexity, followed by a pass that admits only integral results. Later, Bagnara et al. [6] improved this algorithm and presented one that requires $\mathcal{O}(n^3)$.

octagon — i.e., the smallest conjunction of linear inequalities of the above form — that contains $\{m_1, \ldots, m_k\} \subseteq \mathbb{Z}^n$ [166, Sect. 2.3]. With a domain operation $\sqcup_{\mathsf{oct}}$ that joins two octagons, a naïve algorithm (such as Alg. 6) would then enumerate the m_i one after another, represent them as DBMs, and join the DBM representing m_i with the current octagonal abstraction. However, this strategy heavily depends on the number k of models $m_1, \ldots, m_k$, and also the order in which they are examined (cp. Ex. 4.1). We thus present a different strategy. Given a formula φ, our tactic to compute an optimal octagonal abstraction presented herein is to (1) draw a constraint $\pm\langle\!\langle \boldsymbol{v}_1 \rangle\!\rangle \pm \langle\!\langle \boldsymbol{v}_2 \rangle\!\rangle \leq d$ from a fixed set of templates, and (2) find the least value of d subject to φ using dichotomic search. The force of this approach is a predictable (and often smaller) number of calls to a decision procedure. Clearly, if $\boldsymbol{v}_i = (\boldsymbol{v}_i[0], \ldots, \boldsymbol{v}_i[w-1])$, then $\langle\!\langle \boldsymbol{v}_i \rangle\!\rangle \in [-2^{w-1}, 2^{w-1}-1]$ (cp. Cor. 3.1). It follows that $\pm\langle\!\langle \boldsymbol{v}_1 \rangle\!\rangle \pm \langle\!\langle \boldsymbol{v}_2 \rangle\!\rangle \in [-2^w, 2^w]$, which entails that d can be represented as a bit-vector $\boldsymbol{d} = (\boldsymbol{d}[0], \ldots, \boldsymbol{d}[w+1])$. Determining the exact value of $\langle\!\langle \boldsymbol{d} \rangle\!\rangle$ thus amounts to finding the least upper bound of the constraint $\pm\langle\!\langle \boldsymbol{v}_1 \rangle\!\rangle \pm \langle\!\langle \boldsymbol{v}_2 \rangle\!\rangle$. We can thus compute $\langle\!\langle \boldsymbol{d} \rangle\!\rangle$ by applying an interval subdivision scheme similar to Chap. 3.1.2.

Abstracting Linear Template Inequalities

Algorithm 7 presents an implementation for the generalized problem of maximizing a linear expression $\sum_{i=1}^n \lambda_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle$ of n variables $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n$ where $\lambda_1, \ldots, \lambda_n \in \mathbb{Z}$ are constants. The algorithm takes as input a formula φ and a linear expression $\sum_{i=1}^n \lambda_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle$, which is encoded in κ. We assume that this formula is appropriately sign-extended to a bit-width $w' \geq w$ that prevents wraps in κ.

Proposition 4.1. *Rewrite $\sum_{i=1}^n \lambda_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle \leq d$ into the form $\sum_{i=1}^n \lambda_i^+ \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle \leq d + \sum_{i=1}^n \lambda_i^- \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle$ where $(\lambda_1^+, \ldots, \lambda_n^+), (\lambda_1^-, \ldots, \lambda_n^-) \in \mathbb{N}^n$. A propositional encoding of $\sum_{i=1}^n \lambda_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle \leq d$ without wraps is possible with a signed representation in*

$$1 + \lceil \log_2(1 + \max(2^w \cdot (\sum_{i=1}^n \lambda_i^+), b + 2^w \cdot (\sum_{i=1}^n \lambda^-))) \rceil$$

bits.

Proof. To see that such an encoding is possible, assume, without loss of generality, that the disequality is integral and d is non-negative. Rewrite the inequality as $\sum_{i=1}^n \lambda_i^+ \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle \leq d + \sum_{i=1}^n \lambda_i^- \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle$ where $(\lambda_1^+, \ldots, \lambda_n^+), (\lambda_1^-, \ldots, \lambda_n^-) \in \mathbb{N}^n$. Further, let $\lambda^+ = \sum_{i=1}^n \lambda_i^+$ and $\lambda^- = \sum_{i=1}^n \lambda_i^-$. Since $\langle\!\langle \boldsymbol{v}_j \rangle\!\rangle \in [-2^{w-1}, 2^{w-1}-1]$ for each $\boldsymbol{v}_i \in \boldsymbol{V}$, it follows that computing the sums $\sum_{i=1}^n \lambda_i^+ \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle$ and $d + \sum_{i=1}^n \lambda_i^- \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle$ with a signed $1 + \lceil \log_2(1 + \max(2^w \cdot \lambda^+, d + 2^w \cdot \lambda^-)) \rceil$-bit representation is sufficient to avoid wraps, allowing the disequality to be modeled exactly (cp. [75, Sect. 3.3]). □

Hence, $\sum_{i=1}^{n} \lambda_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle$ is expressed over an extended set of bit-vectors $\boldsymbol{W}$ defined as

$$\boldsymbol{W} \quad = \quad \{ \quad (\boldsymbol{v}[0], \ldots, \boldsymbol{v}[w-1], \boldsymbol{v}[w], \ldots, \boldsymbol{v}[w'-1]) \mid \boldsymbol{v} \in \boldsymbol{V} \quad \}$$

where w' respects Prop. 4.1. The supported form of linear expressions includes, most notably, octagonal constraints. The algorithm also supports more expressive classes of linear templates such as octahedra [62], but also less expressive ones such as pentagons [157]. Also note that the algorithm can be seen as a generalization of Alg. 1 to compute interval abstractions, i.e.:

$$\texttt{maximum}(\varphi, \boldsymbol{v}) = \alpha^{\boldsymbol{V}}_{\mathsf{lin-exp}}(\varphi, \langle\!\langle \boldsymbol{v} \rangle\!\rangle) \qquad \texttt{minimum}(\varphi, \boldsymbol{v}) = \alpha^{\boldsymbol{V}}_{\mathsf{lin-exp}}(\varphi, -\langle\!\langle \boldsymbol{v} \rangle\!\rangle)$$

The output of the procedure $\alpha^{\boldsymbol{V}}_{\mathsf{lin-exp}}(\varphi, \sum_{i=1}^{n} \lambda_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle)$ is the least upper bound $d \in \mathbb{Z}$ of $\sum_{i=1}^{n} \lambda_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle$ subject to φ. Lines 3–9 provide special treatment for the sign (since $\neg \boldsymbol{d}[w'-1]$ indicates a positive value of $\langle\!\langle \boldsymbol{d} \rangle\!\rangle$). Then, lines 10–18 represent the core of the algorithm. Since the goal is to maximize $\sum_{i=1}^{n} \lambda_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle$, the algorithm instantiates each remaining bit $\boldsymbol{d}[0], \ldots \boldsymbol{d}[w'-2]$ with true, starting with $\boldsymbol{d}[w'-2]$, and checks satisfiability of the respective formula. If satisfiable, bit $\boldsymbol{d}[j]$ is fixed to true and the output d is incremented by 2^j. Then, the next highest bit is examined. If unsatisfiable, bit $\boldsymbol{d}[j]$ can only take the value false, and $\langle\!\langle \boldsymbol{d} \rangle\!\rangle$ is thus not modified. The algorithm then moves on to maximize the next highest bit. Observe that constraints that express lower bounds, such as $d \leq \sum_{i=1}^{n} \lambda_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle$, can be represented as $\sum_{i=1}^{n} -\lambda_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle \leq -d$. This permits maximization as used in $\alpha^{\boldsymbol{V}}_{\mathsf{lin-exp}}(\varphi, \sum_{i=1}^{n} \lambda_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle)$ to be applied to compute lower bounds, too.

Proposition 4.2. *Let $\varphi \in \wp(\wp(\boldsymbol{V}))$ encode the semantics of a block. Then:*

$$\alpha^{\boldsymbol{V}}_{\mathsf{lin-exp}}(\varphi, \textstyle\sum_{i=1}^{n} \lambda_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle) \quad = \quad \max \left\{ x \in \mathbb{Z} \;\middle|\; \begin{array}{ll} \mathbf{m} : \boldsymbol{V} \to \mathbb{Z} \models \varphi & \wedge \\ x = \sum_{i=1}^{n} \lambda_i \cdot \mathbf{m}(\boldsymbol{v}_i) & \end{array} \right\}$$

Proof. Let $\varphi' = \varphi \wedge \kappa$. Then, φ' encodes the relational semantics φ of the analyzed block augmented with an additional assignment $(\sum_{i=1}^{n} \lambda_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle = \langle\!\langle \boldsymbol{d} \rangle\!\rangle)$. Put $\widehat{\boldsymbol{V}} = \mathsf{vars}(\varphi') \setminus \boldsymbol{V}$. Since the additional variables that constitute φ' appear freely, $\varphi \equiv \exists \widehat{\boldsymbol{V}} : \varphi'$. Intuitively, φ' describes the same relations among $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n$ as φ. Correctness of Alg. 7 thus follows directly from correctness of Alg. 1. □

We present two immediate consequences:

Corollary 4.1. *Let $\varphi \in \wp(\wp(\boldsymbol{V}))$ and $d = \alpha^{\boldsymbol{V}}_{\mathsf{lin-exp}}(\varphi, \sum_{i=1}^{n} \lambda_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle)$. Then, $\varphi \wedge (\sum_{i=1}^{n} \lambda_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle = d)$ is satisfiable.*

Corollary 4.2. *$\alpha^{\boldsymbol{V}}_{\mathsf{lin-exp}}(\varphi, \sum_{i=1}^{n} \lambda_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle)$ requires w' calls to a solver.*

Algorithm 7 $\alpha^{\boldsymbol{V}}_{\mathsf{lin-exp}} : (\wp(\wp(\boldsymbol{V})) \times \wp(\wp(\boldsymbol{W}))) \to \mathbb{Z}$

Input: $\varphi \in \wp(\wp(\boldsymbol{V}))$
Input: linear expression $\sum_{i=1}^{n} \lambda_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle \in \wp(\wp(\boldsymbol{W}))$
Output: least upper bound $d \in \mathbb{Z}$ of $\sum_{i=1}^{n} \lambda_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle$ subject to φ

1: $\kappa \leftarrow \sum_{i=1}^{n} \lambda_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle = \langle\!\langle \boldsymbol{d} \rangle\!\rangle$
2: $\psi \leftarrow \varphi \wedge \kappa$
{check the sign}
3: **if** $\psi \wedge \neg \boldsymbol{d}[w'-1]$ is satisfiable **then**
4: $d \leftarrow 0$
5: $\psi \leftarrow \psi \wedge \neg \boldsymbol{d}[w'-1]$
6: **else**
7: $d \leftarrow -2^{w'-1}$
8: $\psi \leftarrow \psi \wedge \boldsymbol{d}[w'-1]$
9: **end if**
{iterate over bits $w'-2, \ldots, 0$}
10: **for** $i = 1$ to $k-1$ **do**
11: $j \leftarrow w' - i - 1$
{increment d by 2^j if the augmented formula is satisfiable}
12: **if** $\psi \wedge \boldsymbol{d}[j]$ is satisfiable **then**
13: $d \leftarrow d + 2^j$
14: $\psi \leftarrow \psi \wedge \boldsymbol{d}[j]$
15: **else**
16: $\psi \leftarrow \psi \wedge \neg \boldsymbol{d}[j]$
17: **end if**
18: **end for**
19: **return** d

Abstracting Octagons

The iterative application of $\alpha^{\boldsymbol{V}}_{\mathsf{lin-exp}}$ to derive an octagonal abstraction $\alpha^{\boldsymbol{V}}_{\mathsf{oct}}(\varphi)$ is given in Alg. 8. The procedure iterates over all pairs $(\boldsymbol{v}_i, \boldsymbol{v}_j)$ of bit-vectors and all possible combinations (λ_i, λ_j) of coefficients drawn from $\{-1, 1\}$, and then invokes $\alpha^{\boldsymbol{V}}_{\mathsf{lin-exp}}$ for the constraint $\lambda_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle + \lambda_j \cdot \langle\!\langle \boldsymbol{v}_j \rangle\!\rangle \leq d$. Each constraint is thus analyzed separately, one after another. Observe that this formulation does not require $\boldsymbol{v}_i \neq \boldsymbol{v}_j$, and thus represents inequalities $\langle\!\langle \boldsymbol{v}_i \rangle\!\rangle \leq d$ by $\langle\!\langle \boldsymbol{v}_i \rangle\!\rangle + \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle \leq 2 \cdot d$. This representation of range constraints using octagons thus squares with the one proposed by Miné [166, Sect. 2.2]. However, we represent inequalities explicitly rather than using DBMs to support a straightforward formulation of the algorithm.

Example 4.4. *Consider the formula* $\varphi = [\![\texttt{ADD R0 R1}]\!] \wedge [\![m_{\texttt{ADD R0 R1},U}]\!]$ *over inputs* $\boldsymbol{r0}$ *and* $\boldsymbol{r1}$*. Then,* φ *implicitly describes those values of* $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle$ *and* $\langle\!\langle \boldsymbol{r1} \rangle\!\rangle$ *for which*

Algorithm 8 $\alpha^{\boldsymbol{V}}_{\mathsf{oct}} : \wp(\wp(\boldsymbol{V})) \to \mathsf{Oct}$

Input: $\varphi \in \wp(\wp(\boldsymbol{V}))$
Output: least octagon $o \in \mathsf{Oct}$ describing $\boldsymbol{V}$ subject to φ

```
1: o ← ⊤_oct
2: for each pair (v_i, v_j) in V do
3:    for each (λ_i, λ_j) ∈ {−1, 1} × {−1, 1} do
4:       d ← α^V_{lin−exp}(φ, λ_i · ⟨⟨v_i⟩⟩ + λ_j · ⟨⟨v_j⟩⟩)
5:       o ← o ∧ (λ_i · ⟨⟨v_i⟩⟩ + λ_j · ⟨⟨v_j⟩⟩ ≤ d)
6:    end for
7: end for
8: return o
```

an underflow of `ADD R0 R1` *occurs. Applying* $\alpha^{\boldsymbol{V}}_{\mathsf{lin-exp}}$ *to the octagonal constraint* $\langle\langle \boldsymbol{r0} \rangle\rangle + \langle\langle \boldsymbol{r1} \rangle\rangle \leq \langle\langle \boldsymbol{d} \rangle\rangle$ *implicitly extends the constraint to a 10-bit representation so as to prevent wraps, hence* $\boldsymbol{d} = (\boldsymbol{d}[0], \ldots, \boldsymbol{d}[9])$*. First,* $\varphi \wedge \neg \boldsymbol{d}[9]$ *is examined, which turns out unsatisfiable, hence* $-512 \leq \langle\langle \boldsymbol{r0} \rangle\rangle + \langle\langle \boldsymbol{r1} \rangle\rangle \leq -1$ *and* $d = -512$*. In the second iteration, a constraint on* $\boldsymbol{d}[8]$ *is added to give the formula* $\varphi \wedge \boldsymbol{d}[9] \wedge \boldsymbol{d}[8]$*. From satisfiability, we deduce* $-256 \leq \langle\langle \boldsymbol{r0} \rangle\rangle + \langle\langle \boldsymbol{r1} \rangle\rangle \leq -1$ *and* d *is incremented by* 2^8 *to give* $d = -256$*. Then,* $\varphi \wedge \boldsymbol{d}[9] \wedge \boldsymbol{d}[8] \wedge \boldsymbol{d}[7]$ *is unsatisfiable. Since the remaining instances are all satisfiable, we obtain the output* $\langle\langle \boldsymbol{r0} \rangle\rangle + \langle\langle \boldsymbol{r1} \rangle\rangle \leq -129$*. Performing the analysis in its entirety leads to:*

$$\alpha^{\{\boldsymbol{r0},\boldsymbol{r1}\}}_{\mathit{oct}}(\varphi) \;=\; \begin{cases} -128 \;\leq\; \langle\langle \boldsymbol{r0} \rangle\rangle \;\leq\; -1 \\ -128 \;\leq\; \langle\langle \boldsymbol{r1} \rangle\rangle \;\leq\; -1 \\ -256 \;\leq\; \langle\langle \boldsymbol{r0} \rangle\rangle + \langle\langle \boldsymbol{r1} \rangle\rangle \;\leq\; -129 \\ -127 \;\leq\; \langle\langle \boldsymbol{r0} \rangle\rangle - \langle\langle \boldsymbol{r1} \rangle\rangle \;\leq\; 127 \end{cases}$$

It can easily be seen that $\alpha^{\{\boldsymbol{r0},\boldsymbol{r1}\}}_{\mathit{oct}}(\varphi)$ *exactly describes those inputs that indicate an underflow of* `ADD R0 R1`*. The results for all three modes are given in Fig. 4.3.*

Observe that Alg. 8 can be seen as diametrically opposed to Alg. 6. The procedure $\mathtt{sbt}(\varphi, (2^{\mathbb{Z}^n}, \gamma_{\mathsf{oct}}, \alpha_{\mathsf{oct}}, \mathsf{Oct}))$ enumerates solutions and iteratively computes the merge $o \sqcup_{\mathsf{oct}} \alpha_{\mathsf{oct}}(\mathbf{m})$ of an existing octagonal abstraction $o \in \mathsf{Oct}$ with the abstraction of a model $\mathbf{m} : \boldsymbol{V} \to \mathbb{Z}$ so as to converge onto a sound abstraction from below. By way of contrast, the procedure $\alpha^{\boldsymbol{V}}_{\mathsf{oct}}(\varphi)$ starts from $\top_{\mathsf{oct}}$ rather than $\bot_{\mathsf{oct}}$, and incrementally refines intermediate abstractions so as to converge onto the result from above. The following complexity result is an immediate consequence from the solving strategy.

Proposition 4.3. *Let* $\boldsymbol{V} = \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\}$ *with* $\boldsymbol{v}_i = (\boldsymbol{v}_i[0], \ldots, \boldsymbol{v}_i[w-1])$ *and* $\varphi \in \wp(\wp(\boldsymbol{V}))$*. Computing* $\alpha^{\boldsymbol{V}}_{\mathit{oct}}(\varphi)$ *requires*

$$\left(\frac{n \cdot (n-1)}{2}\right) \cdot (w+2) \cdot 8$$

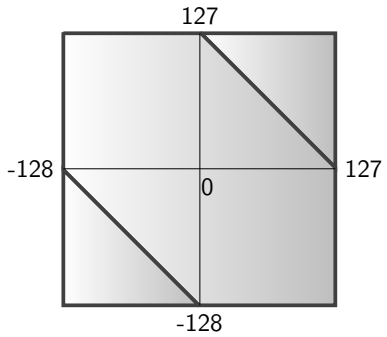


Figure 4.3: Three shaded areas indicate values of $\langle\langle \boldsymbol{x} \rangle\rangle$ and $\langle\langle \boldsymbol{y} \rangle\rangle$ that lead to overflow, underflow, and regular behavior of $\langle\langle \boldsymbol{x} \rangle\rangle + \langle\langle \boldsymbol{y} \rangle\rangle$; redundant constraints, which touch the enclosed volume in a single point, are omitted

calls to a solver.

Proof. From $\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$, we deduce that the overall number of two-variable relations described by an n-dimensional octagon is $\sum_{i=1}^{n-1} i = \frac{n \cdot (n-1)}{2}$. Further, each two-variable relation consists of 8 inequalities. As Alg. 8 iterates over each bit $w+1, \ldots, 0$ in each constraint, we obtain the above result. □

Of course, if range constraints are expressed as $\langle\langle \boldsymbol{v} \rangle\rangle \leq d$ rather than $2 \cdot \langle\langle \boldsymbol{v} \rangle\rangle \leq 2 \cdot d$ and caching is applied to avoid repeated analysis of the same interval constraints, then the overall number of calls can be reduced to:

$$\left(\frac{n \cdot (n-1)}{2} \right) \cdot (w+2) \cdot 4 + 2 \cdot n \cdot w$$

Please note that there is no reason why packing of octagons [166, Sect. 6.2] could not be combined with static dependency analysis [175] to reduce the overall number of constraints that are analyzed.[6] An interesting aspect of the abstraction mechanism $\alpha^{\boldsymbol{V}}_{\mathsf{oct}}(\varphi)$ is that the algorithm yields an octagon that is tightly closed. Intuitively, this means that all hyperplanes defined by $\alpha^{\boldsymbol{V}}_{\mathsf{oct}}(\varphi)$ indeed touch the enclosed volume (see Fig. 4.4).

Definition 4.1. *Let* $^{\bullet} : \mathsf{Oct} \to \mathsf{Oct}$ *denote the tight closure operation [166, Def. 4].*

[6] The key idea of Oh et al. [175] is to statically determine semantic dependencies between different variables, which are then used to partition the variables into different, independent packs. Their technique thus reduces both, memory requirements and computational cost, of octagons and the corresponding domain operations.

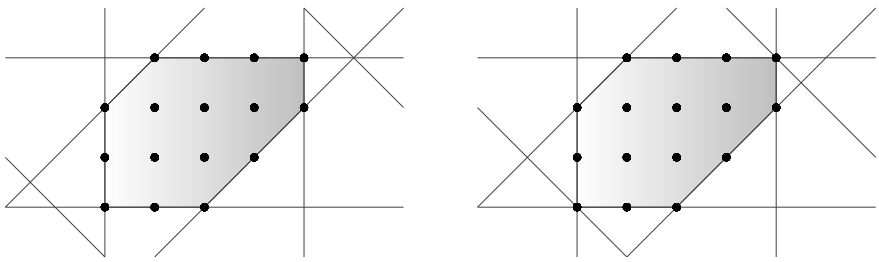


Figure 4.4: Two octagons describe the same space; the octagon on the right-hand side is tightly closed, whereas the octagon on the left is not

Tightly closed octagons can be seen as a normal form representation [166, Sect. 3.5]. The following proposition is a modification of [166, Thm. 3] and [166, Thm. 4] so that it takes integrality of the variables into account.[7]

Proposition 4.4. *Let $o \in \mathsf{Oct}$ over $\boldsymbol{V} = \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\}$. Then, $o = o^\bullet$ iff for all $(\pm\langle\!\langle \boldsymbol{v}_i \rangle\!\rangle \pm \langle\!\langle \boldsymbol{v}_j \rangle\!\rangle \leq d) \in o$ there exists $(x_1, \ldots, x_n) \in \gamma_{\mathsf{oct}}(o)$ such that $\pm x_i \pm x_j = d$.*

The following proposition can be seen as an immediate consequence of Cor. 4.1, Prop. 4.2, and Prop. 4.4.

Proposition 4.5. *Let $\varphi \in \wp(\wp(\boldsymbol{V}))$ denote a formula over $\boldsymbol{V} = \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\}$. Then $\alpha^{\boldsymbol{V}}_{\mathsf{oct}}(\varphi) \in \mathsf{Oct}$ is tightly closed, i.e., $\alpha^{\boldsymbol{V}}_{\mathsf{oct}}(\varphi) = (\alpha^{\boldsymbol{V}}_{\mathsf{oct}}(\varphi))^\bullet$.*

Proof. Let $o = \alpha^{\boldsymbol{V}}_{\mathsf{oct}}(\varphi)$ and assume $o \neq o^\bullet$. Without loss of generality, assume a redundant constraint $\lambda_1 \cdot \langle\!\langle \boldsymbol{v}_1 \rangle\!\rangle + \lambda_2 \cdot \langle\!\langle \boldsymbol{v}_2 \rangle\!\rangle \leq d$, which entails that $(v_1, v_2, \ldots, v_n) \in \gamma^{\boldsymbol{V}}_{\mathsf{oct}}(o)$ such that $\lambda_1 \cdot v_1 + \lambda_2 \cdot v_2 = d$ does not exist, following from Prop. 4.4. Then, $\varphi \wedge (\lambda_1 \cdot \langle\!\langle \boldsymbol{v}_1 \rangle\!\rangle + \lambda_2 \cdot \langle\!\langle \boldsymbol{v}_2 \rangle\!\rangle = d)$ is unsatisfiable, which contradicts Cor. 4.1. □

However, tightly closed octagons can negatively affect the performance of abstract interpretations since closed systems will typically contain many redundant constraints. To improve the efficiency of octagonal abstract interpretations (or guard evaluation in our case), these octagonal systems should be transformed into a so-called *reduced form*, a representation that was proposed by Bagnara et al. [7]. Correctness and optimality of Alg. 8 follows directly from Prop. 4.2, Prop. 4.5, and the structure of the octagon abstract domain.

Corollary 4.3. *Let φ denote a formula and $\boldsymbol{V} = \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\}$. Further, let $G = (\wp(\mathbb{Z}^n), \gamma_{\mathsf{oct}}, \alpha_{\mathsf{oct}}, \mathsf{Oct})$ denote a Galois connection between concrete states and the domain of octagons as defined in [166, Sect. 2.3]. Then $\alpha^{\boldsymbol{V}}_{\mathsf{oct}}(\varphi) = \mathtt{sbt}(\varphi, G)$.*

[7] Observe that Miné [166] assumes non-emptiness of octagons for his algorithms. We omit this detail because only satisfiable formulae φ are abstracted, which entails that $\alpha^{\boldsymbol{V}}_{\mathsf{oct}}(\varphi)$ is non-empty.

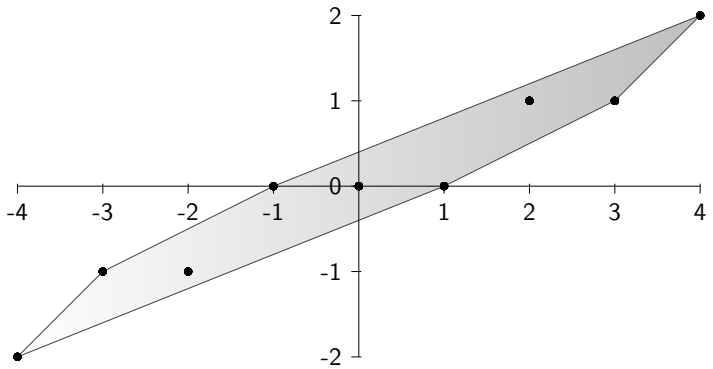


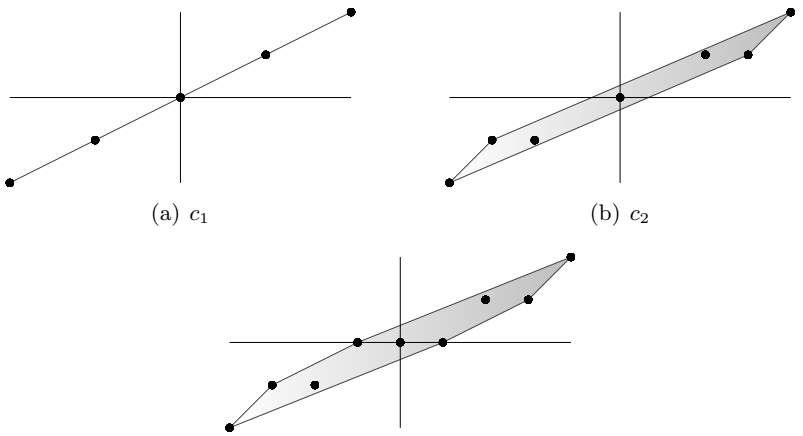
Figure 4.5: Polyhedral abstraction of `y = x div 2` subject to $-4 \leq \mathtt{x} \leq 4$

4.2.2 Convex Polyhedra

In our special setting, the abstract domain of convex polyhedra consists of conjunctions of arbitrary linear inequalities over bit-vectors $\boldsymbol{V} = \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\}$. Although its structure appears — at least to some extent — similar to that of the octagon domain, an abstraction procedure $\alpha^{\boldsymbol{V}}_{\mathsf{conv}}(\varphi)$ for polyhedra differs fundamentally from $\alpha^{\boldsymbol{V}}_{\mathsf{oct}}(\varphi)$. Recall that a key issue in the design of $\alpha^{\boldsymbol{V}}_{\mathsf{oct}}(\varphi)$ is to avoid the join $\sqcup_{\mathsf{oct}}$ of two octagons altogether, which is achieved by analyzing template constraints $\pm\langle\!\langle \boldsymbol{v}_i \rangle\!\rangle \pm \langle\!\langle \boldsymbol{v}_j \rangle\!\rangle \leq d$ one after another. Such a strategy is not possible for convex polyhedra because neither is the number of inequalities known before the analysis, nor are the coefficients in each inequality.

Worked Example

Consider a formula $\varphi \in \wp(\wp(\boldsymbol{V}))$ that encodes the solutions from Fig. 4.5 over $\boldsymbol{V} = \{\boldsymbol{x}, \boldsymbol{y}\}$. From now on, we refer to the domain of convex polyhedra over bit-vectors with a signed interpretation as Conv. Further, let $\sqcup_{\mathsf{conv}} : \mathsf{Conv} \times \mathsf{Conv} \to \mathsf{Conv}$ denote the join of two polyhedra, and $\alpha^{\mathbb{Z}^n}_{\mathsf{conv}} : \wp(\mathbb{Z}^n) \to \mathsf{Conv}$ the convex hull of a set of points. We invoke a solver on φ by applying the minimization/maximization scheme from Alg. 1 to φ in each axis ($\langle\!\langle \boldsymbol{x} \rangle\!\rangle$ and $\langle\!\langle \boldsymbol{y} \rangle\!\rangle$ in this case). However, rather than using the extremal values of $\langle\!\langle \boldsymbol{x} \rangle\!\rangle$ and $\langle\!\langle \boldsymbol{y} \rangle\!\rangle$ separately (which is akin to a box or interval abstraction), we extract entire models $\mathbf{m} : \boldsymbol{V} \to \mathbb{Z}$ that satisfy the respective extremal value. The key idea behind using minimization/maximization is to obtain extremal values which span an extended polyhedron, rather than computing the polyhedral abstraction of a model $\mathbf{m} : \boldsymbol{V} \to \mathbb{Z}$ directly. Intuitively, we aim to find models that represent vertices of the enclosing convex shape, which is advantageous over strategies that enumerate interior points. We further observe that intermediate



(a) c_1 (b) c_2

(c) c_3

Figure 4.6: c_1, c_2, and c_3 after the first three iterations of polyhedral abstraction

results for extremal values provide potential to guide the search for other variables; for instance, if a model $\mathbf{m}_1$ indicates an extremal solution for $\langle\langle \boldsymbol{x} \rangle\rangle$, then a model $\mathbf{m}_2$ for a maximal value of $\langle\langle \boldsymbol{y} \rangle\rangle$ clearly satisfies $\mathbf{m}_2(\boldsymbol{y}) \geq \mathbf{m}_1(\boldsymbol{y})$. Interleaving polyhedral abstraction with minimization/maximization thus dovetails with the desire to converge onto the polyhedral abstraction within few iterations. We obtain two models $\mathbf{m}_1 : \boldsymbol{V} \to \mathbb{Z}$ and $\mathbf{m}_2 : \boldsymbol{V} \to \mathbb{Z}$ defined as $\mathbf{m}_1 = \{\langle\langle \boldsymbol{x} \rangle\rangle = 4, \langle\langle \boldsymbol{y} \rangle\rangle = 2\}$ and $\mathbf{m}_2 = \{\langle\langle \boldsymbol{x} \rangle\rangle = -4, \langle\langle \boldsymbol{y} \rangle\rangle = -2\}$. The polyhedral abstraction of $\mathbf{m}_1$ and $\mathbf{m}_2$, denoted c_1, is defined (see Fig. 4.6(a)):

$$c_1 \;=\; \alpha^{\mathbb{Z}^n}_{\mathsf{conv}}\left(\left\{\begin{array}{l}(\mathbf{m}_1(\boldsymbol{x}), \mathbf{m}_1(\boldsymbol{y})),\\ (\mathbf{m}_2(\boldsymbol{x}), \mathbf{m}_2(\boldsymbol{y}))\end{array}\right\}\right) \;=\; \left\{\begin{array}{lcll}\langle\langle \boldsymbol{y} \rangle\rangle & \geq & -2 & \wedge\\ \langle\langle \boldsymbol{y} \rangle\rangle & \leq & 2 & \wedge\\ \langle\langle \boldsymbol{y} \rangle\rangle & = & \frac{1}{2} \cdot \langle\langle \boldsymbol{x} \rangle\rangle & \end{array}\right\}$$

In the next iteration, we pass $\varphi \wedge \neg c_1$ to a solver to give $\mathbf{m}_3 = \{\langle\langle \boldsymbol{x} \rangle\rangle = -3, \langle\langle \boldsymbol{y} \rangle\rangle = -1\}$ and $\mathbf{m}_4 = \{\langle\langle \boldsymbol{x} \rangle\rangle = 3, \langle\langle \boldsymbol{y} \rangle\rangle = 1\}$. These models are represented as a polyhedron (cp. Fig. 4.6(b)) which is joined with c_1 to give:

$$c_2 \;=\; \left\{\begin{array}{lcllcll}\langle\langle \boldsymbol{y} \rangle\rangle & \leq & \langle\langle \boldsymbol{x} \rangle\rangle + 2 & \wedge & \langle\langle \boldsymbol{y} \rangle\rangle & \geq & \langle\langle \boldsymbol{x} \rangle\rangle - 2 \quad \wedge\\ \langle\langle \boldsymbol{y} \rangle\rangle & \leq & \frac{3}{7} \cdot \langle\langle \boldsymbol{x} \rangle\rangle + \frac{2}{7} & \wedge & \langle\langle \boldsymbol{y} \rangle\rangle & \geq & \frac{3}{7} \cdot \langle\langle \boldsymbol{x} \rangle\rangle - \frac{2}{7}\end{array}\right\}$$

In a third iteration, we pass $\varphi \wedge \neg c_2$ to a solver, giving models $\mathbf{m}_5 = \{\langle\langle \boldsymbol{x} \rangle\rangle = -1, \langle\langle \boldsymbol{y} \rangle\rangle = 0\}$ and $\mathbf{m}_6 = \{\langle\langle \boldsymbol{x} \rangle\rangle = 1, \langle\langle \boldsymbol{y} \rangle\rangle = 0\}$. Again, representing $\mathbf{m}_5$ and $\mathbf{m}_6$ as a

Algorithm 9 $\alpha^{\boldsymbol{V}}_{\mathsf{conv}} : \wp(\wp(\boldsymbol{V})) \to \mathsf{Conv}$

Input: $\varphi \in \wp(\wp(\boldsymbol{V}))$
Output: $c \in \mathsf{Conv}$ such that $\varphi \models c$

1: $c \leftarrow \bot_{\mathsf{conv}}$
2: **while** $\varphi \wedge \neg c$ is satisfiable **do**
3: $\quad p \leftarrow \emptyset$
4: $\quad$ **for** $\boldsymbol{v} \in \boldsymbol{V}$ **do**
5: $\quad\quad \ell \leftarrow \texttt{minimum}(\varphi \wedge \neg c, \boldsymbol{v})$
6: $\quad\quad u \leftarrow \texttt{maximum}(\varphi \wedge \neg c, \boldsymbol{v})$
7: $\quad\quad \mathbf{m}_\ell \leftarrow$ model of $\varphi \wedge (\langle\!\langle \boldsymbol{v} \rangle\!\rangle = \ell)$
8: $\quad\quad \mathbf{m}_u \leftarrow$ model of $\varphi \wedge (\langle\!\langle \boldsymbol{v} \rangle\!\rangle = u)$
9: $\quad\quad p \leftarrow p \cup \{(\mathbf{m}_\ell(\boldsymbol{v}_1), \ldots, \mathbf{m}_\ell(\boldsymbol{v}_n)), (\mathbf{m}_u(\boldsymbol{v}_1), \ldots, \mathbf{m}_u(\boldsymbol{v}_n))\}$
10: $\quad$ **end for**
11: $\quad c \leftarrow c \sqcup_{\mathsf{conv}} \alpha^{\mathbb{Z}^n}_{\mathsf{conv}}(p)$
12: **end while**
13: **return** c

convex polyhedron and joining the abstraction with c_2 yields a new result:

$$c_3 \;=\; \left\{ \begin{array}{lcllcl} \langle\!\langle \boldsymbol{y} \rangle\!\rangle & \leq & \langle\!\langle \boldsymbol{x} \rangle\!\rangle + 2 & \wedge & \langle\!\langle \boldsymbol{y} \rangle\!\rangle & \geq & \langle\!\langle \boldsymbol{x} \rangle\!\rangle - 2 & \wedge \\ \langle\!\langle \boldsymbol{y} \rangle\!\rangle & \leq & \frac{1}{2} \cdot \langle\!\langle \boldsymbol{x} \rangle\!\rangle + \frac{1}{2} & \wedge & \langle\!\langle \boldsymbol{y} \rangle\!\rangle & \geq & \frac{2}{5} \cdot \langle\!\langle \boldsymbol{x} \rangle\!\rangle - \frac{2}{5} & \wedge \\ \langle\!\langle \boldsymbol{y} \rangle\!\rangle & \leq & \frac{4}{3} \cdot \langle\!\langle \boldsymbol{x} \rangle\!\rangle + \frac{4}{3} & \wedge & \langle\!\langle \boldsymbol{y} \rangle\!\rangle & \geq & \frac{1}{2} \cdot \langle\!\langle \boldsymbol{x} \rangle\!\rangle - \frac{1}{2} & \end{array} \right\}$$

Finally, φ augmented with $\neg c_3$ is unsatisfiable, which proves $\varphi \models c_3$. The polyhedron c_3 thus over-approximates φ and the procedure terminates (cp. Fig. 4.6(c)). It is noteworthy that the algorithm enumerates exactly the vertices of c_3.

Algorithm

A formalization of the algorithm is given in Alg. 9. The procedure takes as input $\varphi \in \wp(\wp(\boldsymbol{V}))$ and initializes a polyhedron $c \in \mathsf{Conv}$, which represents the intermediate results, to $\bot_{\mathsf{conv}}$. An outer loop is then executed as long as φ is not entailed by c, which is tested by checking $\varphi \wedge \neg c$ for satisfiability. Within this loop, minimization and maximization of each $\boldsymbol{v} \in \boldsymbol{V}$ is performed subject to $\varphi \wedge \neg c$. All extremal points are collected in $p \subseteq \mathbb{Z}^n$ with $|p| \leq 2 \cdot |\boldsymbol{V}|$, which is eventually joined with c. Computing the join has exponential complexity in the worst case, we thus do not directly join c and p whenever a solution is found, but rather defer the join until all $\boldsymbol{v} \in \boldsymbol{V}$ have been treated.

4.2.3 Non-Optimal Polyhedral Abstraction

Despite the optimized solving strategy based on extremal values, obtaining an optimal polyhedral abstraction may still be prohibitively expensive. As an example, suppose $\varphi \in \wp(\wp(\boldsymbol{V}))$ characterizes models that imply a circular or parabolic shape, thereby inducing hundreds of inequalities. The algorithm enumerates these inequalities one after another, which is very costly in terms of runtime.

Mixing Polyhedra and Octagons

To approach this problem of high computational cost, one could apply the following (sub-optimal) strategy: (1) stop polyhedral abstraction once some threshold (e.g., a timeout) is reached (with intermediate result c such that $\varphi \not\models c$), (2) compute an octagonal abstraction $o = \alpha^{\boldsymbol{V}}_{\mathsf{oct}}(\varphi \wedge \neg c)$, and (3) join c and o. Compared to polyhedral abstraction $\alpha^{\boldsymbol{V}}_{\mathsf{conv}}(\varphi)$, the computation of an octagonal abstraction $\alpha^{\boldsymbol{V}}_{\mathsf{oct}}(\varphi)$ has predictable cost (cp. Prop. 4.3), as the maximum number of SAT calls is bounded through the bit-width and the number of constraints. Then, $c \sqcup_{\mathsf{conv}} o$ constitutes a polyhedral over-approximation of φ with the following ordering in precision:

$$\alpha^{\boldsymbol{V}}_{\mathsf{poly}}(\varphi) \quad \sqsubseteq_{\mathsf{conv}} \quad (c \sqcup_{\mathsf{poly}} o) \quad \sqsubseteq_{\mathsf{conv}} \quad \alpha^{\boldsymbol{V}}_{\mathsf{oct}}(\varphi)$$

Example 4.5. *Assume we stop $\alpha^{\boldsymbol{V}}_{conv}(\varphi)$ of φ as in the worked example after the first iteration, giving c_1 as defined in Fig. 4.6(a). Then,*

$$\alpha^{\boldsymbol{V}}_{oct}(\varphi \wedge \neg c_1) \;=\; \left\{ \begin{array}{rcccl} -3 & \leq & \langle\!\langle \boldsymbol{x} \rangle\!\rangle & \leq & 3 \;\wedge \\ -1 & \leq & \langle\!\langle \boldsymbol{y} \rangle\!\rangle & \leq & 1 \;\wedge \\ -2 & \leq & \langle\!\langle \boldsymbol{x} \rangle\!\rangle - \langle\!\langle \boldsymbol{y} \rangle\!\rangle & \leq & 2 \end{array} \right\}$$

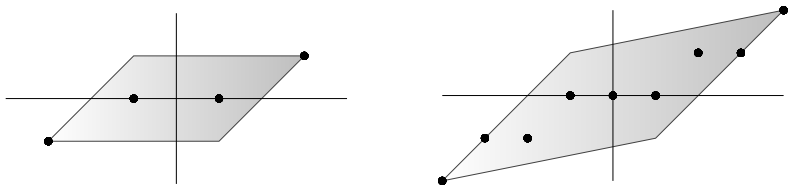
as depicted in Fig. 4.7(a). Then, $c_1 \sqcup_{conv} (\alpha^{\boldsymbol{V}}_{oct}(\varphi \wedge \neg c_1))$, which is given in Fig. 4.7(b), is more precise than $\alpha^{\boldsymbol{V}}_{oct}(\varphi)$, but less precise than $\alpha^{\boldsymbol{V}}_{conv}(\varphi)$ (cp. Fig. 4.6(c)).

Relaxing Inequalities in Template Polyhedra

Combining convex polyhedra with octagons provides one degree of freedom when scaling the computational cost of polyhedral abstraction $\alpha^{\boldsymbol{V}}_{\mathsf{conv}}(\varphi)$. Another direction is to stop polyhedral abstraction prematurely, use an intermediate result of $\alpha^{\boldsymbol{V}}_{\mathsf{conv}}(\varphi)$ to form a template polyhedron [74], and then relax the template inequalities towards a sound abstraction. We illustrate this approach using an example.

Example 4.6. *Consider again the unsound intermediate result c_1 given in Fig. 4.6(a):*

$$c_1 \;=\; \left\{ \begin{array}{rcccl} \langle\!\langle \boldsymbol{y} \rangle\!\rangle & \geq & -2 & \wedge & \langle\!\langle \boldsymbol{y} \rangle\!\rangle - \frac{1}{2} \cdot \langle\!\langle \boldsymbol{x} \rangle\!\rangle \;\leq\; 0 \;\wedge \\ \langle\!\langle \boldsymbol{y} \rangle\!\rangle & \leq & 2 & \wedge & -\langle\!\langle \boldsymbol{y} \rangle\!\rangle + \frac{1}{2} \cdot \langle\!\langle \boldsymbol{x} \rangle\!\rangle \;\leq\; 0 \end{array} \right\}$$



(a) $\alpha^{\boldsymbol{V}}_{\mathsf{oct}}(\varphi \wedge \neg c_1)$ (b) $c_1 \sqcup_{\mathsf{conv}} (\alpha^{\boldsymbol{V}}_{\mathsf{oct}}(\varphi \wedge \neg c_1))$

Figure 4.7: Non-optimal polyhedral abstraction of `y = x div 2` subject to $-4 \leq$ `x` ≤ 4; after the first iteration, polyhedral abstraction is stopped, giving an under-approximation c_1, which is then relaxed using $\alpha^{\boldsymbol{V}}_{\mathsf{oct}}(\varphi \wedge \neg c_1)$

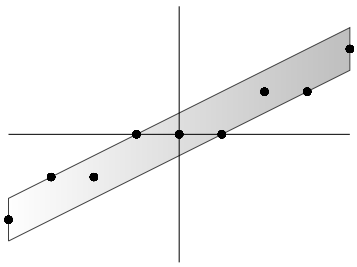


Figure 4.8: Relaxing $\alpha^{\boldsymbol{V}}_{\mathsf{conv}}(\varphi)$ using template constraints

As an alternative to computing $\alpha^{\boldsymbol{V}}_{\mathsf{oct}}(\varphi \wedge \neg c_1)$, we take the left-hand sides of inequalities in c_1 as templates, which are parameterized by constants $d_1, \ldots, d_4$. Transforming the coefficients into integral ones, the template polyhedron has the form:

$$t \;=\; \left\{ \begin{array}{lcllcl} -\langle\!\langle \boldsymbol{y} \rangle\!\rangle & \leq & d_1 & \wedge & 2 \cdot \langle\!\langle \boldsymbol{y} \rangle\!\rangle - \langle\!\langle \boldsymbol{x} \rangle\!\rangle & \leq & d_3 \quad \wedge \\ \langle\!\langle \boldsymbol{y} \rangle\!\rangle & \leq & d_2 & \wedge & -2 \cdot \langle\!\langle \boldsymbol{y} \rangle\!\rangle + \langle\!\langle \boldsymbol{x} \rangle\!\rangle & \leq & d_4 \end{array} \right\}$$

To converge onto a sound polyhedral abstraction of φ starting from c_1, we maximize d_i in each constraint of t separately using $\alpha_{\mathsf{lin-exp}}$ from Alg. 7. This operation gives:

$$t \;=\; \left\{ \begin{array}{lcl} \langle\!\langle \boldsymbol{y} \rangle\!\rangle & \geq & -2 \quad \wedge \\ \langle\!\langle \boldsymbol{y} \rangle\!\rangle & \leq & 2 \quad \wedge \\ 2 \cdot \langle\!\langle \boldsymbol{y} \rangle\!\rangle - \langle\!\langle \boldsymbol{x} \rangle\!\rangle & \leq & 1 \quad \wedge \\ 2 \cdot \langle\!\langle \boldsymbol{y} \rangle\!\rangle - \langle\!\langle \boldsymbol{x} \rangle\!\rangle & \geq & -1 \end{array} \right\} \;=\; \left\{ \begin{array}{lcl} -\langle\!\langle \boldsymbol{y} \rangle\!\rangle & \leq & 2 \quad \wedge \\ \langle\!\langle \boldsymbol{y} \rangle\!\rangle & \leq & 2 \quad \wedge \\ 2 \cdot \langle\!\langle \boldsymbol{y} \rangle\!\rangle - \langle\!\langle \boldsymbol{x} \rangle\!\rangle & \leq & 1 \quad \wedge \\ -2 \cdot \langle\!\langle \boldsymbol{y} \rangle\!\rangle + \langle\!\langle \boldsymbol{x} \rangle\!\rangle & \leq & 1 \end{array} \right\}$$

The result, which features four more integral solutions $(-3, -2)$, $(-1, -1)$, $(1, 1)$, and $(3, 2)$, is given in Fig. 4.8.

A procedure `relax-conv` : $(\wp(\wp(\boldsymbol{V})) \times \mathsf{Conv}) \to \mathsf{Conv}$ that implements this strategy is given in Alg. 10, again based on the assumption that inequalities $\sum_{i=1}^{n} \lambda_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle \leq d$

Algorithm 10 `relax-conv` : $(\wp(\wp(\boldsymbol{V})) \times \mathsf{Conv}) \to \mathsf{Conv}$

Output: $t \in \mathsf{Conv}$ such that $\varphi \models \gamma_{\mathsf{conv}}(o)$

1: $c' \leftarrow$ `make_integral_coefficients`(c)
2: $t \leftarrow \top_{\mathsf{conv}}$
3: **for** each inequality $\sum_{i=1}^{n} \lambda_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle \leq d$ **do**
4: $\quad d \leftarrow \alpha^{\boldsymbol{V}}_{\mathsf{lin-exp}}(\varphi, \sum_{i=1}^{n} \lambda_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle)$
5: $\quad t \leftarrow t \sqcap_{\mathsf{conv}} (\sum_{i=1}^{n} \lambda_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle \leq d)$
6: **end for**
7: **return** t

are appropriately sign-extended to prevent wraps in the expression $\sum_{i=1}^{n} \lambda_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle$. With k template inequalities on input and a length $w' \geq w$ of the sign-extended bit-vectors, the algorithm requires k calls of $\alpha_{\mathsf{lin-exp}}$ which, in turn, requires w' calls to a solver. Procedure `relax-conv` thus calls a solver $k \cdot w'$ times, and the output polyhedron confers the same number of inequalities as the input.

Comparison To conclude the discussion, we observe that relaxation using templates and polyhedral abstraction paired with octagons yield incomparable yet sound results. Hence, if c_1 and c_2 are convex polyhedra obtained using both non-optimal techniques, then neither $c_1 \sqsubseteq_{\mathsf{conv}} c_2$ nor $c_2 \sqsubseteq_{\mathsf{conv}} c_1$. However, with $\alpha^{\boldsymbol{V}}_{\mathsf{conv}}(\varphi) \sqsubseteq_{\mathsf{conv}} c_1$ and $\alpha^{\boldsymbol{V}}_{\mathsf{conv}}(\varphi) \sqsubseteq_{\mathsf{conv}} c_2$, we deduce $\alpha^{\boldsymbol{V}}_{\mathsf{conv}}(\varphi) \sqsubseteq_{\mathsf{conv}} (c_1 \sqcap_{\mathsf{conv}} c_2)$. There is thus no formal argument that precludes combining c_1 and c_2 to obtain a tighter abstraction.

4.2.4 Arithmetical Congruences

Octagons have the ability to describe a limited class of convex shapes, which also can be seen as a form of contiguous ranges of variables. A similar observation holds for convex polyhedra, even though they can express a more general class of geometric shapes. A different class of domain, which determines non-convex strides, is that of arithmetical congruences [114]. Given a bit-vector $\boldsymbol{v} \in \boldsymbol{V}$ with a value set $\{v_1, \ldots, v_k\} \in \mathsf{Val}$, an arithmetical congruence of $\langle\!\langle \boldsymbol{v} \rangle\!\rangle$ is defined by a constant $c \in \mathbb{Z}$ and a modulus $m \in \mathbb{N}$ such that $\{v_1, \ldots, v_n\} \subseteq \{c + k \cdot m \mid k \in \mathbb{Z}\}$. In this case, we shortly write $\langle\!\langle \boldsymbol{v} \rangle\!\rangle \equiv_m c$. Arithmetical congruences have been highlighted as valuable for binary analysis, e.g., for memory access analysis [9, Sect. 3.1].[8]

[8] Note that Balakrishnan and Reps [9, 195] define the abstract domain of strided intervals $m[\ell, u]$, which has the concretization $\{x \mid \ell \leq x \leq u \wedge x = \ell + k \cdot m\}$, rather than considering congruences directly. For example, a strided interval $4[1020, 1028]$ then defines the concrete values $\{1020, 1024, 1028\}$. However, their definition is isomorphic to the reduced product domain of intervals and arithmetical congruences, and can thus also be derived using the techniques presented in this thesis.

Algorithm 11 $\alpha^{\boldsymbol{V}}_{\mathsf{a\text{-}cong}} : \wp(\wp(\boldsymbol{V})) \to \mathsf{A\text{-}Cong}$

Input: $\varphi \in \wp(\wp(\boldsymbol{V}))$
Output: $\langle\!\langle \boldsymbol{v} \rangle\!\rangle \equiv_m c \in \mathsf{A\text{-}Cong}$

```
1:  (m, c) ← (0, ⊥)
2:  while φ is satisfiable do
3:    m ← model of φ
4:    if c = ⊥ then
5:      c ← m(v)
6:    else
7:      v ∈ γ(⟨⟨x⟩⟩ ≡_m c)
8:      d ← abs(m(v) − v)
9:      m ← gcd(m, d)
10:   end if
11:   φ ← φ ∧ ¬(⟨⟨x⟩⟩ ≡_m c)
12: end while
13: c ← c  mod m
14: return  ⟨⟨v⟩⟩ ≡_m c
```

Example 4.7. *To illustrate the benefit of congruences in low-level code analysis, consider* `ASR R0`*, which shifts register* `R0` *to the right by one position. Assume unsigned registers of width* 8*. The instruction has two modes of operation: overflow occurs if the least significant bit of* `R0` *is set on input; otherwise,* `ASR R0` *behaves regularly. Then, abstracting the input* $\boldsymbol{r0}$ *using intervals gives:*

$$\textit{regular} \;:\; \langle \boldsymbol{r0} \rangle \in [0, 254] \qquad \textit{overflow} \;:\; \langle \boldsymbol{r0} \rangle \in [1, 255]$$

By way of comparison, the following arithmetical congruences represent feasible input values of `R0` *in each mode exactly:*

$$\textit{regular} \;:\; \langle \boldsymbol{r0} \rangle \equiv_2 0 \qquad \textit{overflow} \;:\; \langle \boldsymbol{r0} \rangle \equiv_2 1$$

The value sets described by $\langle \boldsymbol{r0} \rangle \equiv_2 0$ *and* $\langle \boldsymbol{r0} \rangle \equiv_2 1$*, respectively, are disjoint, whereas the intervals contain almost as many spurious as legitimate values.*

Algorithm

A dedicated procedure that computes an arithmetical congruence of a bit-vector $\boldsymbol{v} \in \boldsymbol{V}$ subject to $\varphi \in \wp(\wp(\boldsymbol{V}))$ is given in Alg. 11. Its output is the least arithmetical congruence which describes those values of $\langle\!\langle \boldsymbol{v} \rangle\!\rangle$ that satisfy φ. Observe that the algorithm is independent of whether $\boldsymbol{v}$ is interpreted as signed or unsigned; for an

unsigned interpretation, all occurrences of $\langle\!\langle \cdot \rangle\!\rangle$ are replaced by $\langle \cdot \rangle$. First, line 2 initializes the congruence $\langle\!\langle \boldsymbol{v} \rangle\!\rangle \equiv_m c$, which is represented by a pair (m, c), to $(0, \bot)$, representing the infeasible system with $\gamma(\langle\!\langle \boldsymbol{v} \rangle\!\rangle \equiv_m c) = \emptyset$. Then, the algorithm iterates over models of φ. In the first iteration (lines 4 and 5), the algorithm assigns a model $\mathbf{m}(\boldsymbol{v})$ obtained using SAT solving to c, which is interpreted as the displacement of the congruence. Since $m = 0$, $\langle\!\langle \boldsymbol{v} \rangle\!\rangle \equiv_m c$ describes a single value $\{c\}$. Then, φ is restricted to values of $\langle\!\langle \boldsymbol{v} \rangle\!\rangle$ that do not equal c. In the second iteration, the difference d between c and a new model $\mathbf{m}(\boldsymbol{v})$ is computed and $|d|$ is used as the modulus since $\mathtt{gcd}(0, d) = |d|$. Again, a disequality is added to prevent the solutions of $\gamma(\langle\!\langle \boldsymbol{v} \rangle\!\rangle \equiv_m c)$ being found. In the remaining iterations, the statement $m \leftarrow \mathtt{gcd}(m, d)$ refines the modulus (since $\mathtt{gcd}(m, d) < m$) and the loop terminates once an unsatisfiable formula is encountered. Before outputting the result, the representation is simplified using the identity $\langle\!\langle \boldsymbol{v} \rangle\!\rangle \equiv_m c = \langle\!\langle \boldsymbol{v} \rangle\!\rangle \equiv_m (c \mod m)$.

Example 4.8. *Consider φ_R from Ex. 4.7, which describes regular operation of* `ASR R0`*. Suppose* `R0` *is interpreted as unsigned. In the first iteration of Alg. 11, a solver provides $\langle \boldsymbol{r0} \rangle = 4$, hence $(m, c) = (0, 4)$. Then, $\varphi \wedge \neg(\langle \boldsymbol{r0} \rangle \equiv_0 4)$ is passed to the solver, being equivalent to $\varphi \wedge (\langle \boldsymbol{r0} \rangle \neq 4)$. The solver responds with a model that defines $\langle \boldsymbol{r0} \rangle = 8$, which entails $d = 4$ and thus $(m, c) = (4, 4)$. Next, $\varphi \wedge \neg(\langle \boldsymbol{r0} \rangle \equiv_4 4)$ gives $\langle \boldsymbol{r0} \rangle = 10$, from which we compute $(m, c) = (2, 4)$. This result is equivalent to $\langle \boldsymbol{r0} \rangle \equiv_2 4$, which is simplified to $\langle \boldsymbol{r0} \rangle \equiv_2 0$ in line 13 of Alg. 11.*

4.2.5 Affine Equalities

Affine equations [136, 170] are related to linear congruence equations [116, 171, 172], which have the form $\sum_{i=1}^{n} c_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle \equiv_m d$, not to be confused with arithmetical congruences. Here, m denotes the modulus of the linear congruence equation. Indeed, the former domain is a special case of the latter where the modulus m is 0. This suggests adapting an abstraction technique for bit-vector formulae that discovers congruence relationships between the propositional variables of a given formula [145, Fig. 2]. In our setting, however, the problem is different. It is that of computing an affine word-level abstraction of φ defined over bit-vectors $\boldsymbol{V} = \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\}$. As before, we do not aspire to derive relationships that involve intermediate variables, and we assume that each $\boldsymbol{v}_i$ is signed. The algorithm shall eventually compute the affine hull of a set of solutions of $\boldsymbol{V}$ subject to φ [170, Sect. 3].

Definition 4.2. *Without loss of generality, let $G \subseteq \mathbb{Q}^n$. The affine hull of G is defined:*

$$\mathit{aff}(G) \quad = \quad \{\textstyle\sum_{i=1}^{m} \lambda_i \cdot g_i \mid m \geq 1, g_i \in G, \lambda_i \in \mathbb{Q}, \sum_{i=1}^{m} \lambda_i = 1\}$$

Intuitively, $\mathsf{aff}(G)$ corresponds to the affine sub-space generated by a set of points G. Further, Müller-Olm and Seidl [170, Lem. 1] have shown that:

Lemma 4.1. *Let* $\{g_1, \ldots, g_m\} \in \wp(\mathbb{Q}^n)$*. Then,* $\mathit{aff}(\{g_1, \ldots, g_m\}) = \bigsqcup_{i=1}^{n} \{\mathit{aff}(\{g_i\})\}$*.*

Suppose that $\{g_1, \ldots, g_m\} \in \wp(\mathbb{Q}^n)$ denotes all models of φ. The remaining challenge is then to efficiently compute k affine-independent models $h_1, \ldots, h_k \in \mathbb{Q}^n$ with $k \leq m$ such that $\mathsf{aff}(\{h_1, \ldots, h_k\}) = \mathsf{aff}(\{g_1, \ldots, g_m\})$. Such a set $\{h_1, \ldots, h_k\}$ is called an affine basis.

Proposition 4.6. *Let* $\{g_1, \ldots, g_m\} \in \wp(\mathbb{Q}^n)$ *be defined as above. Then, there exists* $\{h_1, \ldots, h_k\} \subseteq \{g_1, \ldots, g_m\}$ *with* $k \leq n + 1$ *such that:*

$$\mathit{aff}(\{h_1, \ldots, h_k\}) \quad = \quad \mathit{aff}(\{g_1, \ldots, g_m\})$$

Proof. Observe that $\mathsf{aff}(G)$ defines an affine sub-space of $\mathbb{Q}^n$. Correctness of the proposition follows from the fact that the height of the domain is $n + 1$, where n is the number of variables, corresponding to the dimension of the induced affine sub-space. Since $(k-1)$-dimensional affine spaces can be represented using k affine-independent points, there exists a smallest (yet not unique) subset $\{h_1, \ldots, h_k\}$ of $\{g_1, \ldots, g_m\}$ that induces the same affine hull as $\{g_1, \ldots, g_m\}$, and therefore, $\mathsf{aff}(\{h_1, \ldots, h_k\}) = \mathsf{aff}(\{g_1, \ldots, g_m\})$ as desired. □

Recall that there exist different approaches to representing affine relations, and even some controversy [170, Sect. 8]. In his seminal work, Karr [136, Sect. 2.1] represented affine spaces as the kernel of an affine transformation, whereas Müller-Olm and Seidl [170, Sect. 1] represent affine spaces using basis vectors. However, our work is concerned with computing transformers over $\boldsymbol{V} = \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\}$, which we interpret as the vector of signed interpretations of the $\boldsymbol{v}_i$, i.e., $\boldsymbol{V} = (\langle\!\langle \boldsymbol{v}_1 \rangle\!\rangle, \ldots, \langle\!\langle \boldsymbol{v}_n \rangle\!\rangle)$. Such a transformation can straightforwardly be represented as $A \cdot \boldsymbol{V} = \mathbf{b}$ for $A \in \mathbb{Q}^{m \times n}$ and $\mathbf{b} \in \mathbb{Q}^n$ or, equivalently, as a matrix $[A \mid \mathbf{b}] \in \mathbb{Q}^{m \times (n+1)}$ (cp. [170, Sect. 2]).

Example 4.9. *Suppose* $G = \{(0,1), (1,2), \ldots, (254, 255)\} \subseteq \mathbb{Q}^2$ *over bit-vectors* $\boldsymbol{V} = (\boldsymbol{v}, \boldsymbol{v}')$*. The affine hull of* G *can be represented as an* 1×3 *matrix as follows:*

$$\mathit{aff}(G) \quad = \quad \left[\begin{array}{cc|c} 1 & -1 & -1 \end{array}\right]$$

An equivalent representation is $\langle\!\langle \boldsymbol{v} \rangle\!\rangle' = \langle\!\langle \boldsymbol{v} \rangle\!\rangle + 1$*. Observe that we also have* $\{(0,1), (1,2)\} \subseteq G$ *and* $\mathit{aff}(\{(0,1), (1,2)\}) = \mathit{aff}(G)$*.*

To keep this chapter self-contained, we formally define the join of two matrices, based on [170, Sect. 3.2] and [144, Sect. 3].

Definition 4.3. *Without loss of generality, let* $[A_1|\mathbf{b}_1] \in \mathbb{Q}^{m_1 \times (n+1)}$ *and* $[A_2|\mathbf{b}_2] \in \mathbb{Q}^{m_2 \times (n+1)}$ *represent two affine systems over* n *variables. With* $I \in \mathbb{Q}^{m \times n}$ *denoting*

the identity matrix, define $[A|\mathbf{b}] \in \mathbb{Q}^{(m_1+m_2+n+1)\times(3\cdot n+3)}$ *as:*

$$[A \mid \mathbf{b}] \;=\; \left[\begin{array}{ccccc|c} 1 & 1 & 0 & 0 & 0 & 1 \\ -\mathbf{b}_1 & 0 & A_1 & 0 & 0 & \\ 0 & -\mathbf{b}_2 & 0 & A_2 & 0 & 0 \\ 0 & 0 & -I & -I & I & 0 \end{array}\right]$$

The join $[A_1|\mathbf{b}_1] \sqcup_{\mathsf{aff}} [A_2|\mathbf{b}_2]$ is then found from $[A \mid \mathbf{b}]$ by (1) triangularizing $[A \mid \mathbf{b}]$ to give $[A' \mid \mathbf{b}']$, and (2) eliminating from $[A' \mid \mathbf{b}']$ those rows for which the first non-zero coefficient is found in the first $2 \cdot n + 2$ columns. Triangularization itself amounts to computing Gaussian elimination.

Example 4.10. *Consider two affine systems* $[A_1|\mathbf{b}_1]$ *and* $[A_2|\mathbf{b}_2]$ *defined as:*

$$[A_1 \mid \mathbf{b}_1] \;=\; \left[\begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 2 \end{array}\right] \qquad [A_2 \mid \mathbf{b}_2] \;=\; \left[\begin{array}{ccc|c} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{array}\right]$$

With Def. 4.3, we put:

$$[A \mid \mathbf{b}] \;=\; \left[\begin{array}{cc|ccc|ccc|ccc|c} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ \hline 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -2 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & 1 & 0 \end{array}\right]$$

Putting A into triangular form to give A' *and eliminating from* A' *all except the last two rows (the leading 7 rows have non-zero coefficients in the first 8 columns) yields:*

$$[A_1|\mathbf{b}_1] \sqcup_{\mathit{aff}} [A_2|\mathbf{b}_2] \;=\; \left[\begin{array}{ccc|c} 2 & 0 & -1 & -2 \\ 0 & 1 & 0 & 0 \end{array}\right]$$

Algorithm

Algorithm 12 gives a technique for computing a word-level abstraction $\alpha^{\boldsymbol{V}}_{\mathsf{aff}}(\varphi)$ of $\boldsymbol{V} = \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\}$ subject to φ. This formulation is equivalent to computing the affine hull $\mathsf{aff}(G) = \bigsqcup_{g \in G} \mathsf{aff}(g)$ of the set $G = \{g \in \mathbb{Q}^n \mid g \models \varphi\}$ of models of φ (cp. [145, Sect. 2]). In what follows, interpret $\boldsymbol{V}$ as the n-ary vector defined as

Algorithm 12 $\alpha^{\boldsymbol{V}}_{\mathsf{aff}}(\varphi) : \wp(\wp(\varphi)) \to \mathsf{Aff}$

```
1:  [A | b] ← [0, ..., 0 | 1]
2:  i ← 0
3:  r ← 1
4:  while i < r do
5:     (a_1, ..., a_n, b_{r-i}) ← row([A | b], r − i)
6:     ψ ← Σ_{i=1}^{n} a_i · ⟨⟨v_i⟩⟩ ≠ b_{r-i}
7:     if φ ∧ ψ is satisfiable then
8:        m ← model of φ ∧ ψ
9:        [A' | b'] ← [A | b] ⊔_aff [Id | (m(v_1), ..., m(v_n))^T]
10:       [A | b] ← triangular([A' | b])
11:       r ← number_of_rows([A | b])
12:    else
13:       i ← i + 1
14:    end if
15: end while
16: return [A | b]
```

$\boldsymbol{V} = (\langle\langle \boldsymbol{v}_1 \rangle\rangle, \ldots, \langle\langle \boldsymbol{v}_n \rangle\rangle)$. Affine equations over $\boldsymbol{V}$ are represented with a matrix $[A \mid \mathbf{b}] \in \mathbb{Q}^{m \times (n+1)}$ where $A \in \mathbb{Q}^{m \times n}$ and $\mathbf{b} = (b_1, \ldots, b_m) \in \mathbb{Q}^m$, which we interpret as defining the set:

$$\{(\langle\langle \boldsymbol{v}_1 \rangle\rangle, \ldots, \langle\langle \boldsymbol{v}_n \rangle\rangle) \in [-2^{w-1}, 2^{w-1} - 1]^n \mid A \cdot \boldsymbol{V} = \mathbf{b}\}$$

The algorithm relies on a propositional encoding for an affine disequality constraint $\sum_{i=1}^{n} a_i \cdot \langle\langle \boldsymbol{v}_i \rangle\rangle \neq b_{r-i}$ where $a_1, \ldots, a_n, b_{r-i} \in \mathbb{Q}$ (cp. Prop. 4.1). In the algorithm, the formula that encodes $\sum_{i=1}^{n} a_i \cdot \langle\langle \boldsymbol{v}_i \rangle\rangle \neq b_{r-i}$ is denoted ψ (see line 6). Apart from ψ, the abstraction algorithm is essentially the same as that proposed for bit-wise linear congruences [145, Fig. 2]. The algorithm starts with an unsatisfiable constraint $\sum_{i=1}^{n} 0 \cdot \langle\langle \boldsymbol{v}_i \rangle\rangle = 1$, which corresponds to $\bot_{\mathsf{aff}}$, the bottom element in the lattice of affine equalities. This constraint is successively relaxed by merging it with a series of affine systems that are derived by SAT (or SMT) solving. The truth assignment $\mathbf{m}$ delivered by the solver is considered to be a map $\mathbf{m} : \boldsymbol{V} \to \mathbb{Z}$. The model $\mathbf{m}$ defines a Boolean value to each propositional variable used in φ. If applied to a w-bit vector of variables such as $\boldsymbol{v} = (\boldsymbol{v}[0], \ldots, \boldsymbol{v}[w-1])$, $\mathbf{m}$ yields a binary vector. Following from Def. 3.2, such a binary vector can then be interpreted as a signed number to give a value in the range $[-2^{w-1}, 2^{w-1} - 1]$. This construction is applied in lines 5-8 to find a vector $(\mathbf{m}(\boldsymbol{v}_1), \ldots, \mathbf{m}(\boldsymbol{v}_n)) \in [-2^{w-1}, 2^{w-1} - 1]^n$ which satisfies both, the disequality $\sum_{i=1}^{n} a_i \cdot \langle\langle \boldsymbol{v}_i \rangle\rangle \neq b_{r-i}$ and the formula φ.

The algorithm is formulated in terms of some auxiliary functions: $\texttt{row}([A|\mathbf{b}], i)$ extracts row i from the matrix $[A|\mathbf{b}]$ where the first row is taken to be row 1;

`triangular`$([A|\mathbf{b}])$ puts $[A \mid \mathbf{b}]$ into an upper triangular form using Gaussian elimination [170]; `number_of_rows`$([A|\mathbf{b}])$ returns the number of rows in $[A|\mathbf{b}]$. The rows of $[A|\boldsymbol{b}]$ are considered in reverse order. Each iteration of the loop tests whether there exists a truth assignment of φ that also satisfies the formula ψ constructed from row $r-i$ of $[A|\mathbf{b}]$. If ψ is unsatisfiable, then every model of φ satisfies the affine equality $\sum_{i=1}^{n} a_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle = b_{r-i}$ represented by row $r-i$ of $[A|\mathbf{b}]$. Hence the equality constitutes a description of the formula. The counter i is then incremented to examine a row which, thus far, has not been considered. Conversely, if the instance is satisfiable, then the solution is represented as a matrix

$$[\mathsf{Id}|(\langle\!\langle \mathbf{m}(\boldsymbol{v}_1) \rangle\!\rangle, \ldots, \langle\!\langle \mathbf{m}(\boldsymbol{v}_n) \rangle\!\rangle)^T] \;=\; \left[\begin{array}{ccccc|c} 1 & 0 & 0 & \ldots & 0 & \mathbf{m}(\boldsymbol{v}_1) \\ 0 & 1 & 0 & \ldots & 0 & \mathbf{m}(\boldsymbol{v}_2) \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & 0 & \ldots & 1 & \mathbf{m}(\boldsymbol{v}_n) \end{array}\right]$$

which is merged with $[A|\boldsymbol{b}]$. Merge is an $O(n^3)$ operation [136], which yields a new summary $[A'|\mathbf{b}']$ that enlarges $[A|\mathbf{b}]$ with the fresh solution. The next iteration of the loop will either relax $[A|\mathbf{b}]$ by finding another solution, or verify that the current row describes φ. Triangular form ensures that all rows beneath the one under consideration are not affected by the merge. At most $n+1$ iterations are required since the affine systems constitute an ascending chain over n variables [136].

Example 4.11. *Consider φ, which encodes $\langle\!\langle \boldsymbol{z} \rangle\!\rangle = 2 \cdot (\langle\!\langle \boldsymbol{v} \rangle\!\rangle + 1) + \langle\!\langle \boldsymbol{y} \rangle\!\rangle$ subject to additional constraints $-32 \leq \langle\!\langle \boldsymbol{v} \rangle\!\rangle \leq 31$ and $-32 \leq \langle\!\langle \boldsymbol{y} \rangle\!\rangle \leq 31$, defined as:*

$$\varphi = \left\{\begin{array}{ll} (\neg \boldsymbol{w}[0]) \wedge \left(\wedge_{i=0}^{6} \boldsymbol{w}[i+1] \leftrightarrow (\boldsymbol{v}[i] \oplus \wedge_{j=0}^{i-1} \boldsymbol{v}[j])\right) & \wedge \\ (\neg \boldsymbol{x}[0]) & \wedge \\ \left(\wedge_{i=0}^{6} \boldsymbol{x}[i+1] \leftrightarrow (\boldsymbol{w}[i] \wedge \boldsymbol{x}[i]) \vee (\boldsymbol{w}[i] \wedge \boldsymbol{y}[i]) \vee (\boldsymbol{x}[i] \wedge \boldsymbol{y}[i])\right) & \wedge \\ \left(\wedge_{i=0}^{7} \boldsymbol{z}[i] \leftrightarrow \boldsymbol{w}[i] \oplus \boldsymbol{x}[i] \oplus \boldsymbol{y}[i]\right) & \wedge \\ ((\boldsymbol{v}[7] \leftrightarrow \boldsymbol{v}[6]) \wedge (\boldsymbol{v}[6] \leftrightarrow \boldsymbol{v}[5])) \wedge ((\boldsymbol{y}[7] \leftrightarrow \boldsymbol{y}[6]) \wedge (\boldsymbol{y}[6] \leftrightarrow \boldsymbol{y}[5])) & \end{array}\right.$$

Suppose $\boldsymbol{v}_1 = \boldsymbol{v}$, $\boldsymbol{v}_2 = \boldsymbol{y}$ and $\boldsymbol{v}_3 = \boldsymbol{z}$ and take φ as input to Alg. 12. Initially, we have $[A \mid \mathbf{m}] = [\ 0 \quad 0 \quad 0 \quad \mid 1] = \bot_{\mathsf{aff}}$ and $r - i = 1$. Then, in the first iteration, ψ corresponds to true*. Passing $\varphi \wedge \psi$ to a solver gives a model $\mathbf{m}_1$, which can be seen as a map from bit-vectors $\{\boldsymbol{v}, \boldsymbol{y}, \boldsymbol{z}\}$ to $\mathbb{Z}$ (cp. line 8 in Alg. 12), e.g., $\mathbf{m}_1 = \{\langle\!\langle \boldsymbol{v} \rangle\!\rangle = 0, \langle\!\langle \boldsymbol{y} \rangle\!\rangle = 0, \langle\!\langle \boldsymbol{z} \rangle\!\rangle = 2\}$. This model is then represented as an affine system $[\mathsf{Id} \mid (\mathbf{m}_1(\boldsymbol{v}), \mathbf{m}_1(\boldsymbol{y}), \mathbf{m}_1(\boldsymbol{z}))^T]$. Joining this system with $\bot_{\mathsf{aff}}$ does not change the result, and triangularization leaves it unchanged, too. We thus have*

$$[A \mid \mathbf{b}] \;=\; \left[\begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 2 \end{array}\right]$$

and $r = 3$. In the second iteration, ψ thus encodes $\langle\!\langle \boldsymbol{z} \rangle\!\rangle \neq 2$ and we pass $\varphi \wedge \psi$ to a solver. Suppose the solver provides a model $\mathbf{m}_2$ for $\varphi \wedge \psi$ defined as $\mathbf{m}_2 = \{\langle\!\langle \boldsymbol{v} \rangle\!\rangle = -1, \langle\!\langle \boldsymbol{y} \rangle\!\rangle = 0, \langle\!\langle \boldsymbol{z} \rangle\!\rangle = 0\}$. As before, this model induces an affine system $[\mathsf{Id} | (\mathbf{m}_2(\boldsymbol{v}), \mathbf{m}_2(\boldsymbol{y}), \mathbf{m}_2(\boldsymbol{z}))^T]$, which is joined with $[A|\mathbf{b}]$ to give:

$$\left[\begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 2 \end{array}\right] \sqcup_{\mathsf{aff}} \left[\begin{array}{ccc|c} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{array}\right] = \left[\begin{array}{ccc|c} 2 & 0 & -1 & -2 \\ 0 & 1 & 0 & 0 \end{array}\right]$$

Then, in the third iteration, ψ encodes $\langle\!\langle \boldsymbol{y} \rangle\!\rangle \neq 0$, and the solver produces:

$$\mathbf{m}_3 = \{\ \boldsymbol{v} \mapsto 0 \quad , \quad \boldsymbol{y} \mapsto 1 \quad , \quad \boldsymbol{z} \mapsto 3\ \}$$

Joining $[\mathsf{Id} \mid (\mathbf{m}_3(\boldsymbol{v}), \mathbf{m}_3(\boldsymbol{y}), \mathbf{m}_3(\boldsymbol{z}))^T]$ with $[A|\mathbf{b}]$ yields $[2\ 1\ -1 \mid -2]$. Further, $\varphi \wedge (2 \cdot \langle\!\langle \boldsymbol{v} \rangle\!\rangle + 2 \cdot \langle\!\langle \boldsymbol{y} \rangle\!\rangle - \langle\!\langle \boldsymbol{z} \rangle\!\rangle \neq -2)$ is unsatisfiable, and the algorithm thus recovers $2 \cdot \langle\!\langle \boldsymbol{v} \rangle\!\rangle + \langle\!\langle \boldsymbol{y} \rangle\!\rangle - \langle\!\langle \boldsymbol{z} \rangle\!\rangle = -2$ from φ. Observe that in this example the unsatisfiable case is first encountered in the final iteration, though this is not always so (in case of affine abstractions that consist of multiple independent equalities).

We conclude this chapter by stating and proving correctness of Alg. 12, showing that the method indeed yields an affine matrix that describes the affine hull of φ.

Lemma 4.2. *Let $\boldsymbol{V} = \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\}$ and $\varphi \in \wp(\wp(\boldsymbol{V}))$. Further, let $G = \{q \in \mathbb{Z}^n \mid q \models \varphi\}$ denote the models of φ. Then, $\mathsf{aff}(G) = \alpha^{\boldsymbol{V}}_{\mathsf{aff}}(\varphi)$.*

Proof. Let $r = \texttt{number_of_rows}([A|\mathbf{b}])$. Let e_j denote the equation $\sum_{i=1}^{n} a_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle = b_{r-i}$ denote the equation in row $r - i$ of $[A|\mathbf{b}]$. On entry and exit of the while loop in Alg. 12, $[A|\mathbf{b}]$ is in triangular form. Further, the following invariant holds:

$$\mathsf{aff}(G) = (\mathsf{aff}(G) \sqcup_{\mathsf{aff}} \{e_j \mid 1 \leq j \leq r - i\}) \cup \{e_j \mid r - i < j \leq r\}$$

This invariant is clearly satisfied immediately before the while loop. Assume some model $\mathbf{m} = (x_1, \ldots, x_n)$ of φ violates e_j. The strongest set of affine equations that satisfies $(x_1, \ldots, x_n)$ as well as every solution to $[A|\mathbf{b}]$ is $(A \cdot \boldsymbol{V} = \mathbf{v}) \sqcup_{\mathsf{aff}} e_j$. Let $[C|\mathbf{c}]$ be the matrix corresponding to $e_1, \ldots, e_{r-i}$. Likewise, let $[E|\mathbf{e}]$ denote the matrix corresponding to $e_{r-i+1}, \ldots, e_r$. Then, $(a_1, \ldots, a_n, b_{r-i})$ is the last row of $[C|\mathbf{c}]$. The model $\mathbf{m} = (x_1, \ldots, x_n)$ satisfies $E \cdot \mathbf{m} = \mathbf{e}$, but not e_{r-i}. Note that $[C|\mathbf{c}]$, $[E|\mathbf{e}]$ and

$$\left[\begin{array}{c|c} C & \mathbf{c} \\ E & \mathbf{e} \end{array}\right]$$

are all in triangular form. The effect of the join $\sqcup_{\mathsf{aff}}$ is to leave $[E \mid \mathbf{e}]$ unchanged and to remove the last row of $[C|\mathbf{c}]$; for each remaining row of $[C|\mathbf{c}]$ the index of the leading entry remains unchanged. Hence, the upper triangular form of

$$\left[\begin{array}{c|c} C & \mathbf{c} \\ E & \mathbf{e} \end{array}\right] \sqcup_{\mathsf{aff}} [\mathsf{Id}|\mathbf{m}]$$

is of the form:

$$\left[\begin{array}{c|c} D & \mathbf{d} \\ E & \mathbf{e} \end{array}\right]$$

The invariant is thus preserved. Otherwise, each model $\mathbf{m}$ of φ satisfies e_{r-i} and i is incremented, which clearly preserves the invariant. On exit from the while loop, the invariant holds together with $i \geq r$, which entails $\mathsf{aff}(G) = \alpha^{\boldsymbol{V}}_{\mathsf{aff}}(\varphi)$ as desired. □

Corollary 4.4. *Let $\boldsymbol{V} = \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\}$ and $\varphi \in \wp(\wp(\boldsymbol{V}))$. Then, $\alpha^{\boldsymbol{V}}_{\mathsf{aff}}(\varphi)$ requires at most $n+1$ calls to a solver.*

4.2.6 Bounded Polynomials

Relaxing linear equalities to non-linear ones provides one degree of freedom for generalizing updates. Polynomial extensions [170, Sect. 6] have been proposed for generalizing linear equality analysis, and there is no reason why this technique cannot be adapted to the problem of abstracting Boolean formulae. A rational polynomial over variables $\boldsymbol{V} = \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\}$ is an equation

$$\textstyle\sum_{i=1}^{|\mathcal{I}|} \left(c_i \cdot \Pi_{j=1}^{n} \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle^{e_{i,j}} \right) \;=\; 0$$

where $e_{i,j} \leq d \in \mathbb{N}$ are exponents and $\mathcal{I}$ is a set of indices. Such a polynomial can be represented by its coefficients $\mathbf{c} = (\mathbf{c}_I)_{I \in \mathcal{I}} \in (\mathbb{Q}^n)^{\mathcal{I}}$. However, bounded polynomials over $\mathbb{Q}$ still form a linear vector space, which suggests to adapt Alg. 12 to capture such equalities. The idea is to augment the block with fresh variables specifically introduced to denote non-linear terms. The terms are drawn from a finite language of templates that includes monomials up to a fixed degree, e.g., $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle^2 \cdot \langle\!\langle \boldsymbol{r1} \rangle\!\rangle$. Using linear combinations of the registers and the templates, bounded polynomials can be represented. It is likewise possible to support exponentials, e.g., involving $2^{\langle \boldsymbol{r0} \rangle}$.

Example 4.12. *Consider the following basic block which computes the location with an offset relative to the start location of a two-dimensional array. Here the registers* `R0` *and* `R1` *represent the row and column coordinates, which are indexed from 0. Register* `R2` *represents row size, and all registers have a signed interpretation:*

```
MUL R0 R2;    ADD R0 R1;
```

We introduce a single non-linear term $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r1} \rangle\!\rangle$. The analysis thus computes affine relations between $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle$, $\langle\!\langle \boldsymbol{r1} \rangle\!\rangle$, $\langle\!\langle \boldsymbol{r2} \rangle\!\rangle$, $\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle$, and $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2} \rangle\!\rangle$.

The polynomial hull of φ with template monomials $\mathcal{S} = \{s_1, \ldots, s_k\}$ is computed using Alg. 13. The algorithm is essentially that presented to compute the affine hull $\alpha^{\boldsymbol{V}}_{\mathsf{aff}}(\varphi)$. However, the key difference is that $[A|\mathbf{b}]$ now ranges over $n+k$ variables $(\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n, s_1, \ldots, s_k)$ to represent bit-vectors as well as monomials. After a solver is

Algorithm 13 $\alpha^{\boldsymbol{V}}_{\mathsf{poly}}(\varphi, \mathcal{S})$

1: $[A \mid \mathbf{b}] \leftarrow [0, \ldots, 0 \mid 1]$
2: $i \leftarrow 0$
3: $r \leftarrow 1$
4: **while** $i < r$ **do**
5: $\quad (a_1, \ldots, a_n, a_{n+1}, \ldots, a_{n+k}, b_{r-i}) \leftarrow \texttt{row}([A \mid \mathbf{b}], r - i)$
6: $\quad \xi \leftarrow (a_1, \ldots, a_n, a_{n+1}, \ldots, a_{n+k}) \cdot \boldsymbol{V} \neq b_{r-i}$
7: $\quad$ **if** $\varphi \wedge \xi$ is satisfiable **then**
8: $\quad\quad \mathbf{m} \leftarrow$ model of $\varphi \wedge \xi$
9: $\quad\quad (p_1, \ldots, p_m) \leftarrow (\texttt{eval}(s_1, \mathbf{m}), \ldots, \texttt{eval}(s_k, \mathbf{m}))$
10: $\quad\quad [A' \mid \mathbf{b}'] \leftarrow [A \mid \mathbf{b}] \sqcup_{\mathsf{poly}} [\mathsf{Id} \mid (\mathbf{m}(\boldsymbol{v}_1), \ldots, \mathbf{m}(\boldsymbol{v}_n), p_1, \ldots, p_m)^T]$
11: $\quad\quad [A \mid \mathbf{b}] \leftarrow \texttt{triangular}([A' \mid \mathbf{b}])$
12: $\quad\quad r \leftarrow \texttt{number_of_rows}([A \mid \mathbf{b}])$
13: $\quad$ **else**
14: $\quad\quad i \leftarrow i + 1$
15: $\quad$ **end if**
16: **end while**
17: **return** $[A \mid \mathbf{b}]$

invoked to find a model (see line 8), concrete values of the monomials are extracted from $\mathbf{m}$ in line 9, which is implemented using an auxiliary function `eval` that evaluates a monomial $s_i \in \mathcal{S}$ based on $\mathbf{m}$. These valuations are then represented within the affine matrix in line 10. Observe that monomials are also encoded in the disequality constraint ξ in line 6. The algorithm thus enumerates models that do not only violate the linear terms in $[A \mid \mathbf{b}]$, but also the non-linear ones. Further, this representation allows us to implement the merge $\sqcup_{\mathsf{poly}}$ of two polynomial systems as the merge $\sqcup_{\mathsf{aff}}$ of affine systems, the intrinsics of which have already been discussed.

Example 4.13. *Let* $\varphi = [\![$`MUL R0 R2; ADD R0 R1`$]\!]$ *and* $\mathcal{S} = \{\langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2} \rangle\!\rangle\}$. *In the first iteration, Alg. 13 presents* φ *to the solver, which gives a model* $\mathbf{m}_1 = \{\langle\!\langle \boldsymbol{r0} \rangle\!\rangle = 2, \langle\!\langle \boldsymbol{r1} \rangle\!\rangle = 4, \langle\!\langle \boldsymbol{r2} \rangle\!\rangle = 3, \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = 10\}$. *Instead of directly representing* $\mathbf{m}_1$ *as an affine system, an auxiliary variable is introduced whose sole purpose is to represent* $s \in \mathcal{S}$. *In this case, we have a single monomial* $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2} \rangle\!\rangle$. *We thus introduce a variable* $p = \mathbf{m}_1(\boldsymbol{r0}) \cdot \mathbf{m}_1(\boldsymbol{r2}) = 6$. *With the variable ordering* $(\boldsymbol{r0'}, \boldsymbol{r0}, \boldsymbol{r1}, \boldsymbol{r2}, p)$ *on columns, we obtain the affine system* $\mathbf{M}_1 = [A_1 | \mathbf{b}_1] \in \mathbb{Z}^{6 \times 5}$ *defined as:*

$$\mathbf{M}_1 \quad = \quad \left[\begin{array}{ccccc|c} 1 & 0 & 0 & 0 & 0 & 10 \\ 0 & 1 & 0 & 0 & 0 & 2 \\ 0 & 0 & 1 & 0 & 0 & 4 \\ 0 & 0 & 0 & 1 & 0 & 3 \\ 0 & 0 & 0 & 0 & 1 & 6 \end{array}\right]$$

Now the procedure proceeds much like before. The formula φ is augmented with the constraint $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2} \rangle\!\rangle \neq 6$.[9] *Passing the augmented formula to a SAT solver yields a new model $\mathbf{m}_2 = \{\langle\!\langle \boldsymbol{r0} \rangle\!\rangle = 3, \langle\!\langle \boldsymbol{r1} \rangle\!\rangle = 4, \langle\!\langle \boldsymbol{r2} \rangle\!\rangle = 8, \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = 28\}$, which implies $p = 24$. We thus represent m_2 as an 6×5 matrix $\mathbf{M}_2$ and compute the merge:*

$$\mathbf{M}_1 \sqcup_{\mathit{aff}} \mathbf{M}_2 = \left[\begin{array}{ccccc|c} 1 & 0 & 0 & 0 & 0 & 10 \\ 0 & 1 & 0 & 0 & 0 & 2 \\ 0 & 0 & 1 & 0 & 0 & 4 \\ 0 & 0 & 0 & 1 & 0 & 3 \\ 0 & 0 & 0 & 0 & 1 & 6 \end{array}\right] \sqcup \left[\begin{array}{ccccc|c} 1 & 0 & 0 & 0 & 0 & 28 \\ 0 & 1 & 0 & 0 & 0 & 3 \\ 0 & 0 & 1 & 0 & 0 & 4 \\ 0 & 0 & 0 & 1 & 0 & 8 \\ 0 & 0 & 0 & 0 & 1 & 24 \end{array}\right]$$
$$= \left[\begin{array}{ccccc|c} 1 & -18 & 0 & 0 & 0 & -26 \\ 0 & 5 & 0 & -1 & 0 & 7 \\ 0 & 0 & 1 & 0 & 0 & 4 \\ 0 & 0 & 0 & 18 & -5 & 24 \end{array}\right]$$

In the following iteration, the formula is thus augmented with $18 \cdot \langle\!\langle \boldsymbol{r2} \rangle\!\rangle - 5 \cdot \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2} \rangle\!\rangle \neq -24$, which then gives another model $\mathbf{m}_3 = \{\langle\!\langle \boldsymbol{r0} \rangle\!\rangle = 2, \langle\!\langle \boldsymbol{r1} \rangle\!\rangle = 2, \langle\!\langle \boldsymbol{r2} \rangle\!\rangle = 2, \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = 6\}$ that entails $p = 4$. Hence, we obtain:

$$(\mathbf{M}_1 \sqcup_{\mathit{aff}} \mathbf{M}_2) \sqcup_{\mathit{aff}} \mathbf{M}_3 = \left[\begin{array}{ccccc|c} 1 & -18 & 0 & 0 & 0 & -26 \\ 0 & 5 & 0 & -1 & 0 & 7 \\ 0 & 0 & 1 & 0 & 0 & 4 \\ 0 & 0 & 0 & 18 & -5 & 24 \end{array}\right] \sqcup \left[\begin{array}{ccccc|c} 1 & 0 & 0 & 0 & 0 & 6 \\ 0 & 1 & 0 & 0 & 0 & 2 \\ 0 & 0 & 1 & 0 & 0 & 2 \\ 0 & 0 & 0 & 1 & 0 & 2 \\ 0 & 0 & 0 & 0 & 1 & 4 \end{array}\right]$$
$$= \left[\begin{array}{ccccc|c} 1 & -18 & -2 & 0 & 0 & -34 \\ 0 & 10 & 1 & -2 & 0 & 18 \\ 0 & 0 & 4 & -18 & 5 & -8 \end{array}\right]$$

By augmenting φ with $4 \cdot \langle\!\langle \boldsymbol{r1} \rangle\!\rangle - 18 \cdot \langle\!\langle \boldsymbol{r2} \rangle\!\rangle + 5 \cdot \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2} \rangle\!\rangle \neq -8$, we generate yet another model $\mathbf{m}_4 = \{\langle\!\langle \boldsymbol{r0} \rangle\!\rangle = 4, \langle\!\langle \boldsymbol{r1} \rangle\!\rangle = 5, \langle\!\langle \boldsymbol{r2} \rangle\!\rangle = 2, \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = 13\}$. Joining $\mathbf{M}_4$ with $(\mathbf{M}_1 \sqcup_{\mathit{aff}} \mathbf{M}_2) \sqcup_{\mathit{aff}} \mathbf{M}_3$ then gives:

$$((\mathbf{M}_1 \sqcup_{\mathit{aff}} \mathbf{M}_2) \sqcup_{\mathit{aff}} \mathbf{M}_3) \sqcup_{\mathit{aff}} \mathbf{M}_4 = \left[\begin{array}{ccccc|c} 12 & -64 & 0 & -70 & 10 & -152 \\ 0 & 64 & -12 & 70 & -23 & 152 \end{array}\right]$$

Proceeding with one more iteration, we obtain:

$$(((\mathbf{M}_1 \sqcup_{\mathit{aff}} \mathbf{M}_2) \sqcup_{\mathit{aff}} \mathbf{M}_3) \sqcup_{\mathit{aff}} \mathbf{M}_4) \sqcup_{\mathit{aff}} \mathbf{M}_5 = \left[\begin{array}{ccccc|c} 1 & 0 & -1 & 0 & -1 & 0 \end{array}\right]$$

Since adding another disequality yields an unsatisfiable system, the equation

$$\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = \langle\!\langle \boldsymbol{r1} \rangle\!\rangle + s = \langle\!\langle \boldsymbol{r1} \rangle\!\rangle + \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2} \rangle\!\rangle$$

characterizes the polynomial input-output relation described by this block.

[9] Encodings for such monomials and polynomials in propositional Boolean logic have been reported by Fuhs et al. [101] in the context of termination analysis. Such constraints are also directly supported by SMT solvers such as Z3 which accept specifications in bit-vector arithmetic [90].

4.3 Flexible Bit-Widths by Extrapolation

If abstractions are inferred for finite bit-vectors, then the bit-width w of the machine architecture manifests itself within the abstractions. As an example, consider the instruction `INC R0`. With registers of width 8, 16, and 32, our technique computes three guarded updates using interval abstraction $\alpha^{\boldsymbol{V}}_{\mathsf{int}}$ and affine abstraction $\alpha^{\boldsymbol{V}}_{\mathsf{aff}}$:

$$\text{8-bit}: \begin{cases} (-2^7 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq 2^7 - 2) & \Rightarrow & (\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + 1) \\ (2^7 - 1 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq 2^7 - 1) & \Rightarrow & (\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = -2^7) \end{cases}$$

$$\text{16-bit}: \begin{cases} (-2^{15} \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq 2^{15} - 2) & \Rightarrow & (\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + 1) \\ (2^{15} - 1 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq 2^{15} - 1) & \Rightarrow & (\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = -2^{15}) \end{cases}$$

$$\text{32-bit}: \begin{cases} (-2^{31} \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq 2^{31} - 2) & \Rightarrow & (\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + 1) \\ (2^{31} - 1 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq 2^{31} - 1) & \Rightarrow & (\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = -2^{31}) \end{cases}$$

These formulae exhibit a high degree in similarity, and can indeed be seen as parametric in the bit-width w:

$$w\text{-bit}: \begin{cases} (-2^{w-1} \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq 2^{w-1} - 2) & \Rightarrow & (\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + 1) \\ (2^{w-1} - 1 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq 2^{w-1} - 1) & \Rightarrow & (\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = -2^{w-1}) \end{cases}$$

This similarity is no coincidence as over- and underflowing behavior often manifests itself in some relation to -2^{w-1} and $2^{w-1} - 1$, respectively. If a bit-vector is then interpreted as a (signed or unsigned) integer, this relation is preserved. The force of this observation is that formulae for, e.g., 5-bit registers are strictly easier to solve than those expressed over 64 bits, which suggests to search for a method that computes guarded updates for small bit-vectors and soundly extrapolates the results to extended bit-vectors. We conjecture that such a technique may be valuable if (1) statements that are inherently difficult to handle by SAT and SMT solvers — such as integer multiplication or division — are part of the analyzed basic block (cp. [148, Chap. 6.3.1]), or (2) many SAT calls are required to converge onto an abstraction. Then, solving for formulae that range over, e.g., 5-bit registers is significantly more efficient than analyzing formulae over 32-bit registers.

Outline To summarize, this chapter presents a technique which identifies constants within the guarded updates, where octagons and affine equalities are handled separately. These constants are then related to the bit-width w using parametric linear templates. The templates themselves are used to compute a symbolic representation of the constants $d \in \mathbb{Z}$ in inequalities such as $\pm\langle\!\langle \boldsymbol{v}_1 \rangle\!\rangle \pm \langle\!\langle \boldsymbol{v}_2 \rangle\!\rangle \leq d$, which is parametrized by the bit-width w. Template relations for both, octagons and affine equalities, are drawn from a set of three template constraints which are presented in Chap. 4.3.1. Afterwards, we discuss how to verify soundness of the induced template for octagons in Chap. 4.3.2 and for affine equalities in Chap. 4.3.3.

4.3.1 Templates for Extrapolation

Octagonal constraints and affine updates are of the form $\pm\langle\langle \boldsymbol{v}_i \rangle\rangle \pm \langle\langle \boldsymbol{v}_j \rangle\rangle \leq d$ and $\boldsymbol{v}' = \sum_{i=1}^{n} c_i \cdot \langle\langle \boldsymbol{v}_i \rangle\rangle + d$, respectively. We conjecture that d in these equations is somehow related to the bit-width w using an equality that can be expressed as a template. Of course, the template has to mimic modular arithmetic and the effects of the bit-width on signed and unsigned interpretations of bit-vectors.

Definition 4.4. *We define template equalities to draw possible symbolic representations of d from:*

$$\begin{array}{lcl} T_1 & = & 2^{w+k} + l \\ T_2 & = & -2^{w+k} + l \\ T_3 & = & l \end{array}$$

We additionally assume $-w \leq k \leq w$.

It is important to note that the template T_3 also subsumes the cases where $c = 2^k + l$ or $c = -2^k + l$, respectively, as both right-hand sides form a constant value. Further, observe that neither of the above templates defines k and l uniquely.

Example 4.14. *Consider the equality $\langle\langle \boldsymbol{r0'} \rangle\rangle = \langle\langle \boldsymbol{r0} \rangle\rangle + \langle\langle \boldsymbol{r1} \rangle\rangle + 2^8$ on an 8-bit machine, which has an analogue $\langle\langle \boldsymbol{r0'} \rangle\rangle = \langle\langle \boldsymbol{r0} \rangle\rangle + \langle\langle \boldsymbol{r1} \rangle\rangle + 2^{16}$ on a 16-bit machine. Our motivation is to symbolically represent 2^8 using $w = 8$, k, and l so as to extrapolate the 16-bit analogue from the 8-bit equality. Using the template T_1, we could express 2^8 using different constants:*

$$(w = 8 \wedge k = 0 \wedge l = 0) \qquad \left(w = 8 \wedge k = -1 \wedge l = 2^7\right)$$

Of course, the first one provides the correct template to extrapolate the 16-bit equality.

Our technique thus proceeds by comparing two relations obtained for small bit-widths, such as 4 and 5 so as to compute unique candidates for each template.

Example 4.15. *Suppose affine equalities $\langle\langle \boldsymbol{r0'} \rangle\rangle = \langle\langle \boldsymbol{r0} \rangle\rangle + \langle\langle \boldsymbol{r1} \rangle\rangle + 8$ and $\langle\langle \boldsymbol{r0'} \rangle\rangle = \langle\langle \boldsymbol{r0} \rangle\rangle + \langle\langle \boldsymbol{r1} \rangle\rangle + 16$ have been obtained for bit-vectors of width 4 and 5, respectively. For the template T_1, we then obtain:*

$$\begin{array}{lrcl} 4\text{-}bit: & \langle\langle \boldsymbol{r0'} \rangle\rangle & = & \langle\langle \boldsymbol{r0} \rangle\rangle + \langle\langle \boldsymbol{r1} \rangle\rangle + (2^{4+k} + l) \\ 5\text{-}bit: & \langle\langle \boldsymbol{r0'} \rangle\rangle & = & \langle\langle \boldsymbol{r0} \rangle\rangle + \langle\langle \boldsymbol{r1} \rangle\rangle + (2^{5+k} + l) \end{array}$$

We thus obtain the equalities $2^{4+k} + l = 8$ and $2^{5+k} + l = 16$. We put

$$\begin{array}{lrcl} & 8 - 2^{4+k} & = & 16 - 2^{5+k} \\ \Leftrightarrow & -2^{4+k} & = & 8 - 2^{5+k} \\ \Leftrightarrow & 2^{5+k} - 2^{4+k} & = & 8 \end{array}$$

and then compute $k = -1$, which entails $l_1 = l_2 = 0$. The candidate for a 32-bit equality based on template T_1 then has the form:

$$32\text{-}bit: \quad \langle\!\langle \boldsymbol{r0}' \rangle\!\rangle \;=\; \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle + 2^{32+k} + l \;=\; \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle + 2^{31}$$

Proposition 4.7. *Without loss of generality, we assume that affine equalities*

$$d^w \;=\; \textstyle\sum_{i=1}^{n} c_i \cdot \langle\!\langle \boldsymbol{v}_i^w \rangle\!\rangle \qquad d^{w+1} \;=\; \textstyle\sum_{i=1}^{n} c_i \cdot \langle\!\langle \boldsymbol{v}_i^{w+1} \rangle\!\rangle$$

for bit-vectors of length w and $w+1$, respectively, have been computed. For each template $T_1, \ldots, T_3$, we can reformulate the imposed constraints as follows:

$$\begin{array}{lrcl} T_1: & 2^{w+k+1} - 2^{w+k} & = & d^{w+1} - d^w \\ T_2: & 2^{w+k} - 2^{w+k+1} & = & d^w - d^{w+1} \\ T_3: & l & = & d^w \end{array}$$

Since $d^w, d^{w+1} \in \mathbb{Z}$ and $-w \leq k \leq w \in \mathbb{Z}$, all three forms of constraints can easily be solved, e.g., by applying logarithmic laws or using a table look-up.

4.3.2 Extrapolation for Octagons

To extrapolate an octagon $o \in \mathsf{Oct}$ derived from two formulae with small bit-width, we introduce an oracle $\Omega : \{1, \ldots, 3\}$ to (randomly or heuristically) choose a template. Further, we assume two octagonal constraints $\lambda_1 \cdot \langle\!\langle \boldsymbol{v}_1^w \rangle\!\rangle + \lambda_2 \cdot \langle\!\langle \boldsymbol{v}_2^w \rangle\!\rangle \leq d^w$ and $\lambda_1 \cdot \langle\!\langle \boldsymbol{v}_1^{w+1} \rangle\!\rangle + \lambda_2 \cdot \langle\!\langle \boldsymbol{v}_2^{w+1} \rangle\!\rangle \leq d^{w+1}$ with $\lambda_1, \lambda_2 \in \{-1, 1\}$. Applying the transformation discussed in Prop. 4.7 then gives a value for the upper bound in the 32-bit case, denoted d^{32}. The remaining question is that of soundness, namely: *Is $\lambda_1 \cdot \langle\!\langle \boldsymbol{v}_1^{32} \rangle\!\rangle + \lambda_2 \cdot \langle\!\langle \boldsymbol{v}_2^{32} \rangle\!\rangle \leq d^{32}$ an abstraction of bit-vectors $\boldsymbol{v}_1^{32}$ and $\boldsymbol{v}_2^{32}$ of length 32?*

Proposition 4.8. *Let $o = \lambda_1 \cdot \langle\!\langle \boldsymbol{v}_1^{32} \rangle\!\rangle + \lambda_2 \cdot \langle\!\langle \boldsymbol{v}_2^{32} \rangle\!\rangle \leq d^{32}$ denote an octagonal constraint derived by applying Prop. 4.7 to T_Ω. Assume the semantics of the 32-bit block is described by a formula φ. Then $\varphi \models o$ iff $\varphi \wedge \neg o$ is unsatisfiable.*

We present a concrete example that highlights the key steps of this construction.

Example 4.16. *Consider the instruction `ADD R0 R1` in overflow mode, encoded for different bit-widths by formulae φ^w. For bit-widths 4 and 5, we compute $\alpha_{\mathsf{oct}}^{\{\boldsymbol{r0}^w, \boldsymbol{r1}^w\}}(\varphi^w)$ as in Alg. 8 to obtain the following guards:*

$$\begin{array}{lrcccl} 4\text{-}bit: & 8 & \leq & \langle\!\langle \boldsymbol{r0}^4 \rangle\!\rangle + \langle\!\langle \boldsymbol{r1}^4 \rangle\!\rangle & \leq & 14 \\ 5\text{-}bit: & 16 & \leq & \langle\!\langle \boldsymbol{r0}^5 \rangle\!\rangle + \langle\!\langle \boldsymbol{r1}^5 \rangle\!\rangle & \leq & 30 \end{array}$$

Suppose Ω gives a candidate template T_1, which we instantiate as:

$$\begin{array}{lrcccl} 4\text{-}bit: & 2^3 & \leq & \langle\!\langle \boldsymbol{r0}^4 \rangle\!\rangle + \langle\!\langle \boldsymbol{r1}^4 \rangle\!\rangle & \leq & 2^4 - 2 \\ 5\text{-}bit: & 2^4 & \leq & \langle\!\langle \boldsymbol{r0}^5 \rangle\!\rangle + \langle\!\langle \boldsymbol{r1}^5 \rangle\!\rangle & \leq & 2^5 - 2 \end{array}$$

By applying Prop. 4.7 we and octagonal candidate constraint for the 32-bit case:

$$32\text{-}bit: \quad 2^{31} \quad \leq \quad \langle\!\langle \boldsymbol{r}\mathbf{0}^{32} \rangle\!\rangle + \langle\!\langle \boldsymbol{r}\mathbf{1}^{32} \rangle\!\rangle \quad \leq \quad 2^{32} - 2$$

Passing

$$\begin{array}{rl} & \varphi^{32} \wedge \neg(2^{31} \leq \langle\!\langle \boldsymbol{r}\mathbf{0}^{32} \rangle\!\rangle + \langle\!\langle \boldsymbol{r}\mathbf{1}^{32} \rangle\!\rangle \leq 2^{32} - 2) \\ = & \varphi^{32} \wedge ((\langle\!\langle \boldsymbol{r}\mathbf{0}^{32} \rangle\!\rangle + \langle\!\langle \boldsymbol{r}\mathbf{1}^{32} \rangle\!\rangle < 2^{31}) \vee (\langle\!\langle \boldsymbol{r}\mathbf{0}^{32} \rangle\!\rangle + \langle\!\langle \boldsymbol{r}\mathbf{1}^{32} \rangle\!\rangle > 2^{32} - 2)) \end{array}$$

to a solver reveals unsatisfiability, which entails that the induced candidate constraint indeed abstracts φ^{32}.

4.3.3 Extrapolation for Affine Equalities

Applying extrapolation to an affine equality $\langle\!\langle \boldsymbol{v}' \rangle\!\rangle = \sum_{i=1}^{n} c_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle + d$ is a straightforward adaptation of the algorithm presented for extrapolating octagons. Indeed, the strategy of applying Prop. 4.7 remains, yet in this case to specify the constant d in the equality $\langle\!\langle \boldsymbol{v}' \rangle\!\rangle = \sum_{i=1}^{n} c_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle + d$.

Proposition 4.9. *Let $\langle\!\langle \boldsymbol{v}' \rangle\!\rangle = \sum_{i=1}^{n} c_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle + d$ denote an affine equality derived by applying Prop. 4.7 to a template T_Ω. Assume the semantics of the block over extended bit-vectors is described by a formula φ. Then:*

$$\varphi \quad \models \quad (\langle\!\langle \boldsymbol{v}' \rangle\!\rangle = \textstyle\sum_{i=1}^{n} c_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle + d$$

iff

$$\varphi \quad \wedge \quad (\langle\!\langle \boldsymbol{v}' \rangle\!\rangle \neq \textstyle\sum_{i=1}^{n} c_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle + d)$$

is unsatisfiable.

Indeed, this soundness criterion coincides with the termination criterion implemented in Alg. 12. Extrapolating an affine equality with a constant d thus amounts to describing d using a template which is parametric in the bit-width, and then checking soundness of the transformation a posteriori; this approach is applicable to polynomial equalities, too. Experimental evidence in Chap. 4.4.8 reveals that it is often possible to find a parametric representation, and that the approach indeed decreases runtimes. The overhead induced by choosing templates and identifying suitable parameters is imperceivable in practice.

4.4 Experiments

We have implemented the abstractions discussed in this section in C++ on top of the Z3 SMT solver [90]. All experiments were performed on a desktop computer equipped with a 3.4 GHz dual-core processor and 4 GB of RAM. The operating system running was WINDOWS 7, and only a single core was used in our experiments.

Table 4.2: Details of benchmarks

block	#instr.	variables			modes	
		$\lvert V_{\mathsf{in}} \rvert$	interm.	$\lvert V_{\mathsf{out}} \rvert$	overall	feasible
ABS	5	1	4	1	18	3
ADD	1	2	0	1	3	3
ADD&ASR	2	2	1	1	6	6
ASR	1	1	0	1	2	2
ASR&ADD	2	1	1	1	4	2
ASR&INC	2	1	1	1	4	2
INC	1	1	0	1	2	2
INC&ABS	6	1	5	1	36	3
INC&ASR	2	1	1	1	4	3
INC&ASR&INC	3	1	2	1	8	5
INC&LSL	2	1	1	1	4	3
ISIGN	7	2	6	1	54	6
LSL	1	1	0	1	2	2
SWAP	3	2	1	2	1	1

4.4.1 Benchmarks

The benchmark set consists of instructions and blocks that implement standard arithmetic operations (such as increment, addition or shift-left) and ones that implement more sophisticated bit-twiddling (such as absolute value computation or the ISIGN function known from the FORTRAN programming language). The key statistics for the different blocks are given in Tab. 4.2. First, the table lists the name of the respective block, followed by the numbers of instructions and variables. Here, column *interm.* provides the number of intermediate bit-vectors that were introduced during static single assignment conversion. Each abstraction mechanism discussed in the following computes abstractions for all input and output registers; the intermediate variables are ignored. The last two columns list the number of overall modes, followed by those that are feasible. We have also executed our implementation of different blocks from the benchmarks of Schlich [207, Sect. 8], but have not come across a sequence of instructions that was semantically more difficult to analyze than ISIGN, which is based on the assembly listing given in Fig. 4.1 (see [32, Sect. 2.2] for further details on this block). Given a basic block, we have computed abstractions of the respective block using either abstraction mechanism presented so far. However, we refrain from giving further results for value set abstraction as this technique was thoroughly evaluated in Chap. 3.3.

4.4.2 Intervals ($\alpha^{V}_{\mathsf{int}}$)

We first report on $\alpha^{\boldsymbol{V}}_{\mathsf{int}}(\varphi)$, the figures of which are given in Tab. 4.3. The table gives results on interval abstraction of the input and output bit-vectors handled separately, as there is no relational dependency to extract. To assess the influence of the bit-width w on the overall runtime, we have expressed each block using bit-vectors of length 8, 16, 32, and 64, respectively. The table shows that our approach generates abstractions in the order of seconds even for 64-bit architectures. Interestingly, the runtime for each SAT instance increases moderately with the bit-width; the number of SAT instances itself depends on the bit-width, too. Since the number of overall instances to be solved is $2 \cdot w \cdot (|\boldsymbol{V}_{\mathsf{in}}| + |\boldsymbol{V}_{\mathsf{out}}|) \cdot f + o$, where o and f denote the numbers of overall modes and feasible modes, respectively, the moderate increase can be explained solely by the efficiency of the Z3 SMT solver.

4.4.3 Octagons ($\alpha^{V}_{\mathsf{oct}}$)

To compute relational characterizations of the input and output bit-vectors, we applied to the more expensive procedure $\alpha^{\boldsymbol{V}}_{\mathsf{oct}}(\varphi)$. Here, we computed octagonal abstractions for all combinations of input and output bit-vectors. For the example `SWAP`, e.g., which has 2 input registers and 2 output registers, this entails that we have computed abstractions for all 6 combinations of variables. The experimental results are given in Tab. 4.4. Again, we observe that the cost of increasing the bit-width for each SAT instance is moderate, whereas the sheer number of instances has a significant impact on the runtime. For instance, computing an octagon that describes the relation between the two inputs of `ASR&ADD` in all six feasible modes necessitates 9378 SAT instances to be solved for the 64-bit case, requiring 8.48 seconds overall. By way of comparison, an abstraction for 8 bits is derived in 0.25 seconds using 1314 calls to the solver. However, these figures suggest Z3 is able to solve approximately 1000 SAT instances per second in the 64-bit case, depending on the complexity of the block itself.

4.4.4 Convex Polyhedra ($\alpha^{V}_{\mathsf{conv}}$)

Compared to octagonal abstraction $\alpha^{\boldsymbol{V}}_{\mathsf{oct}}(\varphi)$, runtimes for polyhedral abstraction $\alpha^{\boldsymbol{V}}_{\mathsf{conv}}(\varphi)$ vary strongly. Whereas $\alpha^{\boldsymbol{V}}_{\mathsf{oct}}(\varphi)$ evaluates a fixed number of SAT instances that is determined through the numbers of (feasible) mode combinations and registers (as well as their bit-widths), the situation is different for convex polyhedra. Then, the number of iterations depends on the number of hyperplanes described by φ. For some benchmarks, polyhedral abstraction is significantly faster than octagonal abstraction. For example, polyhedral abstraction of an `ADD` instruction stabilizes in less than 0.5 seconds even in the 64-bit case, compared to 3.51 seconds for octagons. However, if φ describes a non-linear shape (such as the multiplication of

Table 4.3: Experimental results for $\alpha^{\boldsymbol{V}}_{\mathsf{int}}(\varphi)$

block	#bits	#SAT	time	block	#bits	#SAT	time
ABS	8	114	0.05	INC & ABS	8	132	0.05
	16	210	0.06		16	228	0.08
	32	402	0.11		32	420	0.48
	64	786	0.39		64	804	4.24
ADD	8	147	0.04	INC & ASR	8	52	0.03
	16	291	0.07		16	100	0.04
	32	579	0.16		32	196	0.08
	64	1155	0.70		64	388	0.20
ADD & ASR	8	294	0.05	INC & ASR & INC	8	248	0.04
	16	582	0.13		16	488	0.07
	32	1058	0.30		32	968	0.11
	64	2310	1.34		64	1928	0.38
ASR	8	66	0.02	INC & LSL	8	100	0.03
	16	130	0.03		16	196	0.07
	32	258	0.06		32	388	0.17
	64	514	0.23		64	772	0.47
ASR & ADD	8	294	0.09	ISIGN	8	342	0.09
	16	582	0.13		16	630	0.12
	32	1058	0.14		32	1206	0.32
	64	2310	0.39		64	2358	1.25
ASR & INC	8	68	0.04	LSL	8	66	0.02
	16	132	0.06		16	130	0.06
	32	260	0.07		32	258	0.08
	64	516	0.37		64	514	0.25
INC	8	66	0.05	SWAP	8	65	0.05
	16	130	0.05		16	129	0.09
	32	258	0.08		32	257	0.13
	64	514	0.20		64	513	0.37

Table 4.4: Experimental results for $\alpha^{\boldsymbol{V}}_{\mathsf{oct}}(\varphi)$

block	#bits	#SAT	time	block	#bits	#SAT	time
ABS	8	234	0.05	INC & ABS	8	252	0.05
	16	426	0.17		16	444	0.11
	32	810	0.76		32	796	0.44
	64	1578	4.43		64	1564	1.06
ADD	8	657	0.14	INC & ASR	8	220	0.04
	16	1233	0.52		16	412	0.11
	32	2385	2.26		32	796	0.44
	64	4689	3.51		64	1564	1.06
ADD & ASR	8	1168	0.20	INC & ASR & INC	8	368	0.06
	16	2192	0.61		16	688	0.30
	32	4240	2.35		32	1328	0.44
	64	8336	7.46		64	2608	1.78
ASR	8	146	0.03	INC & LSL	8	220	0.04
	16	274	0.11		16	412	0.06
	32	530	0.14		32	796	0.39
	64	1042	0.73		64	1564	0.96
ASR & ADD	8	1314	0.25	ISIGN	8	486	0.13
	16	2466	0.94		16	870	0.45
	32	4770	3.81		32	1638	1.41
	64	9378	8.48		64	3174	3.13
ASR & INC	8	148	0.04	LSL	8	146	0.03
	16	276	0.09		16	274	0.10
	32	532	0.20		32	530	0.25
	64	1044	0.88		64	1042	0.72
INC	8	146	0.06	SWAP	8	438	0.16
	16	274	0.08		16	822	0.39
	32	530	0.48		32	1590	1.74
	64	1042	4.24		64	3126	10.69

two 32-bit integers), $\alpha^{\boldsymbol{V}}_{\mathsf{conv}}(\varphi)$ typically requires significantly more calls to the solver than $\alpha^{\boldsymbol{V}}_{\mathsf{oct}}(\varphi)$, frequently leading to timeouts (after 5 minutes). We thus applied two different heuristics to accelerate polyhedral abstraction:

Mixing Polyhedra and Octagons When polyhedral abstraction is mixed with octagons, we preempted $\alpha^{\boldsymbol{V}}_{\mathsf{conv}}(\varphi)$ after 1 second and then combined the intermediate polyhedron with an octagonal abstraction. With this strategy, a polyhedral abstraction was obtained within less than 10 seconds for either benchmark.

Relaxing Inequalities Here, we set the threshold to 16 inequalities within $\alpha^{\boldsymbol{V}}_{\mathsf{conv}}(\varphi)$, upon which polyhedral abstraction stopped. The inequalities were then taken as templates and relaxed using dichotomic search, giving runtimes of less than 8 seconds for each benchmark.

Both heuristics delivered more precise approximations than $\alpha^{\boldsymbol{V}}_{\mathsf{oct}}(\varphi)$, which we measured by enumerating all models of the resulting abstraction.

4.4.5 Arithmetical Congruences ($\alpha^{\boldsymbol{V}}_{\mathsf{a\text{-}cong}}$)

Analysis of arithmetical congruences as presented in Chap. 4.2.4 does, in contrast to the other abstraction procedures, not apply an optimized search strategy. This is because the algorithm relies on a reduction of the modulus in each iteration based on the greatest common divisor of (1) solutions that are already described by an intermediate arithmetical congruence and (2) a newly found solution, which ensures rapid convergence as shown in Tab 4.5. Interestingly, the number of solutions required to compute an abstraction varies with differing bit-widths, although unpredictably. To illustrate, consider a formula over a bit-vector $\boldsymbol{x}$ with models $\langle \boldsymbol{x} \rangle = 0, \langle \boldsymbol{x} \rangle = 2, \ldots, \langle \boldsymbol{x} \rangle = 254$. In the 8-bit case, the solver may first produce $\langle \boldsymbol{x} \rangle = 0$ and $\langle \boldsymbol{x} \rangle = 8$ before $\langle \boldsymbol{x} \rangle = 2$ is found, which then induces the abstraction $\langle \boldsymbol{x} \rangle \equiv_2 0$. By way of contrast, in the 16-bit case, the solver may provide $\langle \boldsymbol{x} \rangle = 0$ and $\langle \boldsymbol{x} \rangle = 2$, and these two models directly entail the desired abstraction. In practice, however, the effect on the runtime is negligible, which may be explained by the fact that strengthening constraints smoothly integrates with incremental SAT solving.

4.4.6 Affine Equalities ($\alpha^{\boldsymbol{V}}_{\mathsf{aff}}$)

The results of our implementation for affine abstraction $\alpha^{\boldsymbol{V}}_{\mathsf{aff}}$ of input-output relations for the benchmarks are given in Tab. 4.6. The impact of the join $\sqcup_{\mathsf{aff}}$ of two affine systems is imperceivable due to the low number of variables and the independence from the bit-width. Observe that the same number of calls to the solver are required for either bit-width, which leads to almost equal runtimes for 8- and

Table 4.5: Experimental results for $\alpha^{V}_{\mathsf{a\text{-}cong}}(\varphi)$

block	#bits	#SAT	time	block	#bits	#SAT	time
ABS	8	37	0.03	INC & ABS	8	55	0.04
	16	37	0.03		16	59	0.03
	32	36	0.06		32	73	0.11
	64	37	0.07		64	60	0.11
ADD	8	33	0.02	INC & ASR	8	22	0.06
	16	29	0.05		16	31	0.07
	32	45	0.14		32	43	0.09
	64	43	0.15		64	45	0.11
ADD & ASR	8	70	0.06	INC & ASR & INC	8	39	0.06
	16	54	0.09		16	39	0.07
	32	85	0.13		32	39	0.10
	64	74	0.13		64	39	0.11
ASR	8	15	0.02	INC & LSL	8	19	0.04
	16	17	0.03		16	25	0.05
	32	16	0.06		32	24	0.14
	64	16	0.06		64	24	0.17
ASR & ADD	8	69	0.04	ISIGN	8	93	0.04
	16	89	0.08		16	104	0.05
	32	74	0.12		32	114	0.09
	64	78	0.18		64	106	0.12
ASR & INC	8	13	0.02	LSL	8	16	0.02
	16	14	0.02		16	17	0.02
	32	14	0.06		32	16	0.03
	64	14	0.07		64	16	0.05
INC	8	10	0.01	SWAP	8	17	0.03
	16	11	0.02		16	11	0.04
	32	10	0.02		32	15	0.10
	64	11	0.03		64	17	0.12

64-bit architectures. We have also applied the technique to several AVR 8-bit microcontroller programs, which have been in the past been used to evaluate the effectiveness of the [MC]SQUARE model checker [207, 208]. The sizes of these programs range from 148 to 289 instructions. Computing affine equalities for all basic blocks amounts to less than 3 seconds for each of these programs.

On Large Coefficients

The affine join $\sqcup_{\mathsf{aff}}$ may lead to excessively large coefficients that can significantly degrade performance. Indeed, this is a well-known problem for linear relations analysis [221], and often necessitates the use of libraries that support multiple precision integers. In our analysis, it is interesting to observe that the size of the coefficients strongly depends on the order in which models are found. To illustrate, consider an instruction `ADD R0 R1` applied to 32-bit registers (in regular mode). Suppose a solver produces two affine systems defined as follows:

$$[A_1|\mathbf{b}_1] = \left[\begin{array}{ccc|c} 1 & 0 & 0 & -100023 \\ 0 & 1 & 0 & 100024 \\ 0 & 0 & 1 & 1 \end{array}\right] \qquad [A_2|\mathbf{b}_2] = \left[\begin{array}{ccc|c} 1 & 0 & 0 & -100024 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & -100023 \end{array}\right]$$

Joining $[A_1|\mathbf{b}_1]$ and $[A_2|\mathbf{b}_2]$ gives:

$$\left[\begin{array}{ccc|c} 100023 & -1 & 0 & -10004700553 \\ 26091 & 26496076796883 & 6072291101674310584 & 8722534784596035683 \end{array}\right]$$

It is important to appreciate, though, that a joined system would be much simpler, had the solver first found $[A_3|\mathbf{b}_3]$ rather than $[A_2|\mathbf{b}_2]$:

$$[A_3|\mathbf{b}_3] = \left[\begin{array}{ccc|c} 1 & 0 & 0 & -100024 \\ 0 & 1 & 0 & 100023 \\ 0 & 0 & 1 & -1 \end{array}\right]$$

Then, $[A_1|\mathbf{b}_1] \sqcup_{\mathsf{aff}} [A_3|\mathbf{b}_3]$ is given as:

$$\left[\begin{array}{ccc|c} 1 & -1 & 0 & -200047 \\ 0 & 2 & -1 & 200047 \end{array}\right]$$

Based on this observation, we have implemented an algorithm that enumerates models in the proximity of solutions found before, which can be encoded straightforwardly using range constraints. Sign-extensions using randomly enumerated models are often in excess of 512 bits, which significantly degrades performance of SAT solving. Proximity-based models, which are likely to induce small coefficients, and thus require only slightly extended bit-vectors, significantly improve performance.

Table 4.6: Experimental results for $\alpha^{\boldsymbol{V}}_{\mathsf{aff}}(\varphi)$

block	#bits	#SAT	time	block	#bits	#SAT	time
ABS	8	27	0.02	INC & ABS	8	45	0.05
	16		0.04		16		0.08
	32		0.05		32		0.10
	64		0.06		64		0.11
ADD	8	15	0.02	INC & ASR	8	13	0.01
	16		0.05		16		0.04
	32		0.06		32		0.04
	64		0.6		64		0.05
ADD & ASR	8	30	0.01	INC & ASR & INC	8	23	0.01
	16		0.06		16		0.05
	32		0.10		32		0.05
	64		0.14		64		0.09
ASR	8	16	0.01	INC & LSL	8	13	0.01
	16		0.02		16		0.02
	32		0.02		32		0.02
	64		0.04		64		0.04
ASR & ADD	8	30	0.02	ISIGN	8	72	0.03
	16		0.07		16		0.04
	32		0.08		32		0.04
	64		0.10		64		0.6
ASR & INC	8	10	0.01	LSL	8	8	0.01
	16		0.02		16		0.02
	32		0.04		32		0.02
	64		0.04		64		0.03
INC	8	8	0.03	SWAP	8	6	0.03
	16		0.03		16		0.04
	32		0.04		32		0.05
	64		0.04		64		0.05

Our implementation restricts models to a cube in the neighborhood of the model found before; in our experiments, we found that a distance of 16 per bit-vector was sufficient. To illustrate the impact of this choice, a naïve implementation of $\alpha^{\boldsymbol{V}}_{\mathsf{aff}}$ for `ABS` expressed over 64 bits requires more than 2 seconds, whereas our implementation terminates within 0.17 seconds. The results for other benchmarks are similar.

4.4.7 Polynomial Equalities ($\alpha^{\boldsymbol{V}}_{\mathsf{poly}}$)

The results for polynomial analysis are given in Tab. 4.7, which also contains the template monomials $\mathcal{S}$. Hence, the overall equality ranges over $\#\mathbf{vars}+|\mathcal{S}|$ variables (the intermediate variables are omitted), thereby directly providing an upper bound on the number of iterations. The benchmarks differ from those used before in that they consists of blocks that multiply bit-vectors, e.g., to index a two-dimensional array or to square an integer. We manually enforced regular behavior for all involved operations. In the most demanding block `SQUARE&MUL`, which requires the longest time, Z3 spends most time trying to prove soundness of the abstraction $\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = \langle\!\langle \boldsymbol{r0} \rangle\!\rangle^2 \cdot \langle\!\langle \boldsymbol{r1} \rangle\!\rangle$ by testing $\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle \neq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle^2 \cdot \langle\!\langle \boldsymbol{r1} \rangle\!\rangle$ for unsatisfiability.

4.4.8 Extrapolation

For all benchmarks, extrapolation from small bit-vectors produced the same results as abstracting the full semantics of, e.g., a 64-bit block. This result confirms our intuition that a numerical abstractions of certain mode combinations often manifests themselves in an exponential relation to the bit-width. To illustrate the impact on runtimes, consider the benchmark `ISIGN` over 64 bits, for which $\alpha^{\boldsymbol{V}}_{\mathsf{oct}}$ is required to solve 3174 instances in 3.13 seconds (cp. Tab. 4.4). For extrapolation, we computed $\alpha^{\boldsymbol{V}}_{\mathsf{oct}}(\varphi)$ for registers of width 4 and 5 respectively, which induced a parametric representation of the octagonal constants, in 0.09 and 0.11 seconds. Then, checking for soundness amounts to solving a single SAT instance for each constraint, which amounts to 0.28 and 0.31 seconds overall (including abstractions for small bit-widths) for 32 and 64 bits, respectively. These figures indicate a reduction of approximately 90% in the 64-bit case. The overall results for `ABS` and `ISIGN` are given in Tab. 4.8. For the simpler benchmarks, the differences in runtime are not as significant. A noteworthy aspect of extrapolation-based abstraction is that the problem of dealing with huge integer coefficients in intermediate results is eliminated altogether.

4.5 Discussion

Thus far, we have presented a variety of techniques to abstract Boolean formulae, the structure of which is unknown, using different classes of abstract domains. An overview of these techniques and their estimated precision and cost are given

Table 4.7: Experimental results for $\alpha^{\boldsymbol{V}}_{\mathsf{poly}}(\varphi, \mathcal{S})$

block	#instr.	#vars	$\mathcal{S}$	#bits	#SAT	time
MUL	1	3	$\langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r1} \rangle\!\rangle$	8	5	0.02
				16		0.03
				32		0.05
				64		0.13
MUL&ADD	2	3	$\langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r1} \rangle\!\rangle$	8	5	0.02
				16		0.03
				32		0.05
				64		0.12
SQUARE	1	2	$\langle\!\langle \boldsymbol{r0} \rangle\!\rangle^2$	8	4	0.02
				16		0.03
				32		0.06
				64		0.11
SQUARE&ADD	2	3	$\langle\!\langle \boldsymbol{r0} \rangle\!\rangle^2$	8	5	0.03
				16		0.06
				32		0.20
				64		0.33
SQUARE&MUL	2	3	$\langle\!\langle \boldsymbol{r0} \rangle\!\rangle^2$ $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r1} \rangle\!\rangle$ $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle^2 \cdot \langle\!\langle \boldsymbol{r1} \rangle\!\rangle$	8	7	0.02
				16		0.08
				32		0.26
				64		0.42

Table 4.8: Octagonal extrapolation for ABS and ISIGN

block	#bits	time
ABS	4	0.06
	5	0.09
	32	0.24
	64	0.24

block	#bits	time
ISIGN	4	0.09
	5	0.11
	32	0.28
	64	0.31

in Tab. 4.9. For all domains, we have presented techniques that yield optimal approximations in the respective domain. The domain of convex polyhedra has the highest cost in the worst case, mostly due to the sheer number of inequalities, which is unsurprising. For this case, we have presented two non-optimal approximation strategies. The first one mixes polyhedral abstraction with octagonal abstraction, whereas the second one uses an intermediate (unsound) abstraction as a template. Yet, it is noteworthy that sometimes $\alpha^{\boldsymbol{V}}_{\mathsf{conv}}(\varphi)$ outperforms $\alpha^{\boldsymbol{V}}_{\mathsf{oct}}(\varphi)$. This is for the same reason for which analysis using reduced octagons outperforms analysis using tight ones [7], i.e., $\alpha^{\boldsymbol{V}}_{\mathsf{conv}}(\varphi)$ enumerates the vertices of the convex polyhedron, whereas $\alpha^{\boldsymbol{V}}_{\mathsf{oct}}(\varphi)$ systematically attempts to maximize each constraint from a fixed set of templates, which may contain redundant ones. If the number of vertices of a polyhedral abstraction is small compared to the number of inequalities of an octagonal abstraction, then $\alpha^{\boldsymbol{V}}_{\mathsf{conv}}(\varphi)$ typically outperforms $\alpha^{\boldsymbol{V}}_{\mathsf{oct}}(\varphi)$.

Polyhedral Abstraction and Widening The goal of both non-optimal techniques for convex polyhedra is to accelerate convergence, a problem which is typically tackled using widenings [79] in the abstract interpretation framework. Formally, a widening operator $\nabla : (\mathsf{Conv} \times \mathsf{Conv}) \to \mathsf{Conv}$ needs to satisfy:

1. $\forall x, y \in \mathsf{Conv} : x \sqsubseteq_{\mathsf{conv}} (x \nabla y)$
2. $\forall x, y \in \mathsf{Conv} : y \sqsubseteq_{\mathsf{conv}} (x \nabla y)$
3. for all increasing chains $x_0 \sqsubseteq_{\mathsf{conv}} x_1 \sqsubseteq_{\mathsf{conv}} \ldots$, the increasing chain $y_0 \sqsubseteq_{\mathsf{conv}} y_1 \sqsubseteq_{\mathsf{conv}} \ldots$ defined by $y_0 = x_0$ and $y_{i+1} = y_i \nabla x_{i+1}$ is ultimately stationary.

A classical approach to widening of convex polyhedra is to eliminate unstable inequalities [82], which can straightforwardly be integrated with our methods. However, it is our belief that more sophisticated techniques such as lookahead widening [113] or widening with landmarks [222] could likewise be combined with our method. The combination and evaluation of automatic abstraction using SAT and SMT solving with widening operators remains a direction for future research.

Table 4.9: Overview of abstractions presented in Chap. 3 and Chap. 4

Domain	**Chap.**	**Precision**	**Cost**
Int	3.1.2	optimal	low
Val	3.1.2	optimal	low – medium
Oct	4.2.1	optimal	medium
Conv	4.2.2	optimal	medium – high
Conv + Oct	4.2.3	non-optimal	medium – high
Conv + `relax-conv`	4.2.3	non-optimal	medium – high
A-Cong	4.2.4	optimal	low
Aff	4.2.5	optimal	low
Poly	4.2.6	optimal	low – medium

5 Transformers for Template Constraints

Abstracting a formula $\varphi \in \wp(\wp(\boldsymbol{V}))$ using one of the abstraction procedures introduced previously provides one way to summarize states or relations between variables as described by a basic block. A vexing question beyond abstraction of direct relations between variables, however, is how symbolic states on input to the block are related to symbolic states on output. To illustrate, suppose a symbolic interval $[\mathtt{R0}_\ell, \mathtt{R0}_u]$ on input of an instruction `ADD R0 R1`. The question then is how symbolic extremal values $\mathtt{R0}'_\ell$ and $\mathtt{R0}'_u$ on output can be characterized using $\mathtt{R0}_\ell$, $\mathtt{R0}_u$, $\mathtt{R1}_\ell$, and $\mathtt{R1}_u$. The desired abstraction in the case of intervals (for 8-bit registers) is:

$$\begin{array}{lcl}
(-128 \leq \mathtt{R0} + \mathtt{R1} \leq 127) & \Rightarrow & \left(\begin{array}{lcll} \mathtt{R0}'_\ell & = & \mathtt{R0}_\ell + \mathtt{R1}_\ell & \wedge \\ \mathtt{R0}'_u & = & \mathtt{R0}_u + \mathtt{R1}_u & \end{array}\right) \\
(-256 \leq \mathtt{R0} + \mathtt{R1} \leq -129) & \Rightarrow & \left(\begin{array}{lcll} \mathtt{R0}'_\ell & = & 256 + \mathtt{R0}_u + \mathtt{R1}_u & \wedge \\ \mathtt{R0}'_u & = & 256 + \mathtt{R0}_\ell + \mathtt{R1}_\ell & \end{array}\right) \\
(128 \leq \mathtt{R0} + \mathtt{R1} \leq -254) & \Rightarrow & \left(\begin{array}{lcll} \mathtt{R0}'_\ell & = & -256 + \mathtt{R0}_\ell + \mathtt{R1}_\ell & \wedge \\ \mathtt{R0}'_u & = & -256 + \mathtt{R0}_u + \mathtt{R1}_u & \end{array}\right)
\end{array}$$

Applying such a guarded update to interval analysis then amounts to intersecting the inputs with the guard and applying the respective update. Algorithms to compute such relations between symbolic constraints, based on blocks that do not range over these symbolic constraints but over concrete variables, are the topic of this chapter. To summarize, this chapter provides a collection of methods for the analysis of symbolic constraints that could be categorized as follows:

Lifting Assuming that an input-output relation between bit-vectors is described using an affine or polynomial equality, techniques that apply lifting exploit the syntactic structure of the respective equality to obtain a relation between intervals or octagons on input and output of a basic block. As an example, assume an affine equality $\langle\!\langle \boldsymbol{v}' \rangle\!\rangle = -\langle\!\langle \boldsymbol{v} \rangle\!\rangle$ abstracts a block. Further, assume symbolic boundaries $\boldsymbol{v}_\ell$, $\boldsymbol{v}_u$, $\boldsymbol{v}'_\ell$, and $\boldsymbol{v}'_u$ such that $\langle\!\langle \boldsymbol{v}_\ell \rangle\!\rangle \leq \langle\!\langle \boldsymbol{v} \rangle\!\rangle \leq \langle\!\langle \boldsymbol{v}_u \rangle\!\rangle$ and $\langle\!\langle \boldsymbol{v}'_\ell \rangle\!\rangle \leq \langle\!\langle \boldsymbol{v}' \rangle\!\rangle \leq \langle\!\langle \boldsymbol{v}'_u \rangle\!\rangle$. An affine system

$$\langle\!\langle \boldsymbol{v}'_\ell \rangle\!\rangle \;=\; -\langle\!\langle \boldsymbol{v}_u \rangle\!\rangle \qquad \langle\!\langle \boldsymbol{v}'_u \rangle\!\rangle \;=\; \langle\!\langle \boldsymbol{v}_\ell \rangle\!\rangle$$

over symbolic intervals can then be derived directly from the equality, without loss of information. Although computationally cheap and easy to implement, techniques that simply lift affine or polynomial equalities over bit-vectors to relational domains such as octagons incur a loss of precision.

Quantification Given an logical characterization $\varphi \in \wp(\wp(\boldsymbol{V}))$ of a basic block, φ can be augmented with additional constraints and quantifiers to specify the relation between symbolic constraints on the bit-level within the formula itself. Quantifiers then need to be eliminated before abstraction, which yields a quantifier-free formula that specifies a direct bit-level relationship between symbolic constraints. These techniques yield optimal abstractions and are, as we think, algorithmically elegant, but suffer from the computational cost of quantifier elimination, even more so since quantifiers appear alternately.

Interleaved Abstraction As an alternative approach, which in terms of cost lies in between techniques based on lifting and quantification, we propose to interleave abstraction using different abstract domains with model enumeration. The key idea in these techniques is to extract relational abstractions d and d' (e.g., using octagons) that describe input and output variables from a concrete model of φ, and then compute an affine or polynomial equality that describes how φ transforms d into d'.

In the following, we sometimes denote a transformer which relates elements of a source domain D_1 to elements of a target domain D_2 using a relational domain T of the transformer by $D_1 \xrightarrow{T} D_2$. Since we assume complete lattices D_1 and D_2 as base domains, monotone functions of type $D_1 \xrightarrow{T} D_2$ themselves form a complete lattice.

Outline We build towards these different techniques and their instances for different abstract domains as follows. First, Chap. 5.1 discusses lifting techniques that are applied to affine and polynomial equalities. Specifically, we discuss lifting to derive the following types of transformers: $\mathsf{Int} \xrightarrow{\mathsf{Aff}} \mathsf{Int}$, $\mathsf{Oct} \xrightarrow{\mathsf{Aff}} \mathsf{Oct}$, $\mathsf{Int} \xrightarrow{\mathsf{Poly}} \mathsf{Int}$, and $\mathsf{Oct} \xrightarrow{\mathsf{Poly}} \mathsf{Oct}$. Lifting affine equalities to intervals does not incur a loss in precision; unfortunately, this is not so for octagons. The following Chap. 5.2 details the quantification-based technique that characterizes symbolic relations $D_1 \xrightarrow{\mathsf{Bool}} D_2$ directly on the bit-level. There, we show how such formulae can be derived so that the source and target domains D_1 and D_2 are given as parameters of the formula. Computing a transformer using, e.g., affine equalities then amounts to affine abstraction of the formula after quantifier elimination. Then, Chap. 5.3 shows techniques that interleave dichotomic search with abstraction. For the case of affine updates on octagons, i.e., $\mathsf{Oct} \xrightarrow{\mathsf{Aff}} \mathsf{Oct}$, this technique yields optimal abstractions. Finally, Chap. 5.4 studies how polyhedral abstraction can applied to derive approximate transformers for intervals and octagons, in case that the underlying basic block can neither be represented using affine equalities nor using polynomials. This chapter concludes with experimental results in Chap. 5.5, a survey of related work in Chap. 5.6 and a discussion in Chap. 5.7.

5.1 Lifting Equalities to Template Domains

The techniques presented thus far are all concerned with relating the values of concrete variables used in a program. Suppose an affine equality $\langle\!\langle \boldsymbol{v}' \rangle\!\rangle = \sum_{i=1}^{n} c_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle + c$ has been derived. Such equalities can then be used, e.g., to compute the value set of $\langle\!\langle \boldsymbol{v}' \rangle\!\rangle$ from value sets of $\langle\!\langle \boldsymbol{v}_1 \rangle\!\rangle, \ldots, \langle\!\langle \boldsymbol{v}_n \rangle\!\rangle$ on input, simply by computing $\sum_{i=1}^{n} c_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle$ for all feasible inputs. As an alternative, affine relations analysis would apply existential quantifier elimination to compute affine invariants. Common to both techniques is that they relate concrete variables.

In abstract interpretation using intervals, however, it is desirable to express how extremal values $\langle\!\langle \boldsymbol{v}'_\ell \rangle\!\rangle$ and $\langle\!\langle \boldsymbol{v}'_u \rangle\!\rangle$ of $\boldsymbol{v}'$ are related to the extremal values $\langle\!\langle \boldsymbol{v}_{1,\ell} \rangle\!\rangle, \langle\!\langle \boldsymbol{v}_{1,u} \rangle\!\rangle, \ldots, \langle\!\langle \boldsymbol{v}_{n,\ell} \rangle\!\rangle, \langle\!\langle \boldsymbol{v}_{n,u} \rangle\!\rangle$ of $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n$ on entry, as opposed to extracting values of $\langle\!\langle \boldsymbol{v}'_\ell \rangle\!\rangle$ and $\langle\!\langle \boldsymbol{v}'_u \rangle\!\rangle$ from the value sets induced by the affine relations. In this section, we show how to systematically derive equalities of the form

$$\begin{array}{rcl} \langle\!\langle \boldsymbol{v}'_\ell \rangle\!\rangle & = & \sum_{i=1}^{n} c_{i,\ell} \cdot \langle\!\langle \boldsymbol{v}_{i,\ell} \rangle\!\rangle + \sum_{i=1}^{n} c_{i,u} \cdot \langle\!\langle \boldsymbol{v}_{i,u} \rangle\!\rangle + c \\ \langle\!\langle \boldsymbol{v}'_u \rangle\!\rangle & = & \sum_{i=1}^{n} d_{i,\ell} \cdot \langle\!\langle \boldsymbol{v}_{i,\ell} \rangle\!\rangle + \sum_{i=1}^{n} d_{i,u} \cdot \langle\!\langle \boldsymbol{v}_{i,u} \rangle\!\rangle + c \end{array}$$

from a given affine equality $\langle\!\langle \boldsymbol{v}' \rangle\!\rangle = \sum_{i=1}^{n} c_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle + c$. We refer to this technique as *lifting*. However, we do not limit ourselves to the specific case $\mathsf{Int} \xrightarrow{\mathsf{Aff}} \mathsf{Int}$, but also study the more general cases of lifting affine and polynomial relations to octagonal (or other linear template) constraints.

5.1.1 Lifting Affine Equalities to Intervals

In this section, we explore how to transform an affine system such as

$$[A|\mathbf{b}] \quad = \quad \left[\begin{array}{cccc|c} 1 & 0 & 1 & 1 & -2^{32} \\ 0 & 1 & 0 & -1 & 0 \end{array}\right]$$

over bit-vectors $\boldsymbol{V} = \{\boldsymbol{r0}, \boldsymbol{r1}, \boldsymbol{r0}', \boldsymbol{r1}'\}$ to interval updates. This transformation consists of lifting the affine abstraction $[A|\mathbf{b}]$, which ranges over variables in $\boldsymbol{V}$, to symbolic range boundaries. To do so, let $\boldsymbol{V}_{\mathsf{in}} = \{\boldsymbol{r0}, \boldsymbol{r1}\} \subseteq \boldsymbol{V}$ denote the bit-vectors on entry of the block described by φ. Likewise, let $\boldsymbol{V}_{\mathsf{out}} = \{\boldsymbol{r0}', \boldsymbol{r1}'\} \subseteq \boldsymbol{V}$ denote the bit-vectors on exit. We introduce sets of fresh variables

$$\begin{array}{rclcrcl} \boldsymbol{V}^{\ell}_{\mathsf{in}} & = & \{\boldsymbol{r0}_\ell, \boldsymbol{r1}_\ell\} & \quad & \boldsymbol{V}^{u}_{\mathsf{in}} & = & \{\boldsymbol{r0}_u, \boldsymbol{r1}_u\} \\ \boldsymbol{V}^{\ell}_{\mathsf{out}} & = & \{\boldsymbol{r0}'_\ell, \boldsymbol{r1}'_\ell\} & & \boldsymbol{V}^{u}_{\mathsf{out}} & = & \{\boldsymbol{r0}'_u, \boldsymbol{r1}'_u\} \end{array}$$

to represent symbolic boundaries of each bit-vector in $\boldsymbol{V}$. If necessary, we transform the equations such that the left-hand side of each equation consists of only one variable in $\boldsymbol{V}_{\mathsf{out}}$, which can be achieved by reordering the columns of the affine system and performing triangularization. For the above system $[A|\mathbf{b}]$, this gives:

$$\langle\!\langle \boldsymbol{r0}' \rangle\!\rangle \quad = \quad -\langle\!\langle \boldsymbol{r0} \rangle\!\rangle - \langle\!\langle \boldsymbol{r1} \rangle\!\rangle - 2^{32} \qquad \langle\!\langle \boldsymbol{r1}' \rangle\!\rangle \quad = \quad \langle\!\langle \boldsymbol{r1} \rangle\!\rangle$$

These equations entail the following affine equalities on symbolic interval boundaries:

$$\begin{array}{rclcrcl} \langle\!\langle \boldsymbol{r0}'_\ell \rangle\!\rangle & = & -\langle\!\langle \boldsymbol{r0}_u \rangle\!\rangle - \langle\!\langle \boldsymbol{r1}_u \rangle\!\rangle - 2^{32} & \qquad & \langle\!\langle \boldsymbol{r1}'_\ell \rangle\!\rangle & = & \langle\!\langle \boldsymbol{r1}_\ell \rangle\!\rangle \\ \langle\!\langle \boldsymbol{r0}'_u \rangle\!\rangle & = & -\langle\!\langle \boldsymbol{r0}_\ell \rangle\!\rangle - \langle\!\langle \boldsymbol{r1}_\ell \rangle\!\rangle - 2^{32} & & \langle\!\langle \boldsymbol{r1}'_u \rangle\!\rangle & = & \langle\!\langle \boldsymbol{r1}_u \rangle\!\rangle \end{array}$$

To derive such a system, transform each of the original equations into the form

$$\boldsymbol{v}' \;=\; \textstyle\sum_{\boldsymbol{v} \in \boldsymbol{V}_{\mathsf{in}}} \lambda_{\boldsymbol{v}} \cdot \boldsymbol{v} + c$$

where $\boldsymbol{v}' \in \boldsymbol{V}_{\mathsf{out}}$, $c \in \mathbb{Q}$, and $\lambda_{\boldsymbol{v}} \in \mathbb{Q}$ for all $\boldsymbol{v} \in \boldsymbol{V}_{\mathsf{in}}$ (observe that, without loss of generality, we assume the right-hand side of the equality to be rational rather than integral); with a suitable variable ordering, this can be achieved. Correctness of this transformation follows from the fact that (1) an affine equation system describes an affine sub-space and (2) invariance of the elementary row operations, namely multiply a row by a non-zero scalar, add a row to another, or permute two rows (see, e.g., [136, Sect. 2] or [115, Sect. 2.2]). Both affine systems thus describe the same sets of solutions.

Example 5.1. *The system below on the left can be transformed into the system on the right by applying elementary row operations:*

$$\left[\begin{array}{cccc|c} 1 & -1 & 0 & 0 & 1 \\ 0 & 1 & 0 & -1 & 2 \end{array}\right] \;\rightsquigarrow\; \left[\begin{array}{cccc|c} 1 & 0 & 0 & -1 & 3 \\ 0 & 1 & 0 & -1 & 2 \end{array}\right]$$

To formalize the lifting transformation, we introduce a map β that replaces a variable $\boldsymbol{v}$ with its symbolic upper bound $\boldsymbol{v}_u$ in case its sign in the affine equality is positive; otherwise, β replaces $\boldsymbol{v}$ with its lower bound $\boldsymbol{v}_\ell$.

Definition 5.1. $\beta : (\mathbb{Q} \times \boldsymbol{V}_{\mathsf{in}}) \to (\boldsymbol{V}^\ell_{\mathsf{in}} \cup \boldsymbol{V}^u_{\mathsf{in}})$ *is defined as:*

$$\beta(\lambda, \boldsymbol{v}) \;=\; \begin{cases} \langle\!\langle \boldsymbol{v}_\ell \rangle\!\rangle & : \textit{ if } \lambda < 0 \\ \langle\!\langle \boldsymbol{v}_u \rangle\!\rangle & : \textit{otherwise} \end{cases}$$

We then substitute each transformed equation $\langle\!\langle \boldsymbol{v}' \rangle\!\rangle = \sum_{\boldsymbol{v} \in \boldsymbol{V}_{\mathsf{in}}} \lambda_{\boldsymbol{v}} \cdot \langle\!\langle \boldsymbol{v} \rangle\!\rangle + c$ by a pair of equations over symbolic range boundaries.

Proposition 5.1. *Consider an equality* $\langle\!\langle \boldsymbol{v}' \rangle\!\rangle = \sum_{\boldsymbol{v} \in \boldsymbol{V}_{\mathsf{in}}} \lambda_{\boldsymbol{v}} \cdot \langle\!\langle \boldsymbol{v} \rangle\!\rangle + c$*. Put:*

$$\begin{array}{rcll} \langle\!\langle \boldsymbol{v}'_\ell \rangle\!\rangle & = & \sum_{\boldsymbol{v} \in \boldsymbol{V}_{\mathsf{in}}} \lambda_{\boldsymbol{v}} \cdot \beta(\lambda_{\boldsymbol{v}}, \boldsymbol{v}) + c & \wedge \\ \langle\!\langle \boldsymbol{v}'_u \rangle\!\rangle & = & \sum_{\boldsymbol{v} \in \boldsymbol{V}_{\mathsf{in}}} \lambda_{\boldsymbol{v}} \cdot \beta(-\lambda_{\boldsymbol{v}}, \boldsymbol{v}) + c & \end{array}$$

The key idea when constructing the upper bound is to replace each occurrence of a variable in the original system with its upper bound in case its coefficient is positive, and with its lower bound otherwise. However, although the presented procedure is sound, fast, and easy-to-implement, it is incomplete. We illustrate incompleteness by means of an example.

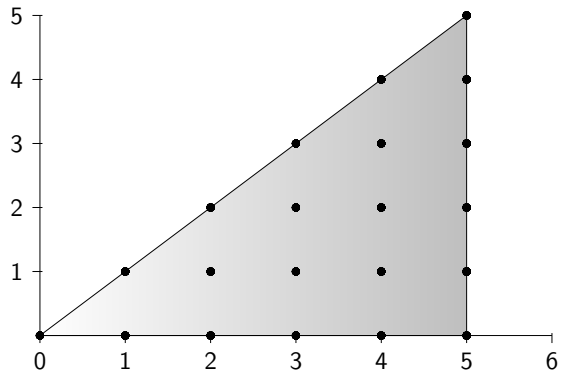


Figure 5.1: Incompleteness of affine lifting

Example 5.2. *Let φ encode a formula with solutions depicted in Fig. 5.1. Since the relation between $\boldsymbol{x}$ and $\boldsymbol{y}$ is non-affine, $\alpha_{\mathsf{aff}}^{\{\boldsymbol{x},\boldsymbol{y}\}}(\varphi) = \top_{\mathsf{aff}}$, and lifting also yields $\top$. However, the optimal abstraction of φ using intervals is $\langle\langle \boldsymbol{y}_\ell \rangle\rangle = 0 \wedge \langle\langle \boldsymbol{y}_u \rangle\rangle = \langle\langle \boldsymbol{x}_u \rangle\rangle$.*

Yet, lifting as performed in Prop. 5.1 is still complete relative to the affine equality on input. In Lem. 4.2, we have shown optimality of affine abstraction $\alpha_{\mathsf{aff}}^{\boldsymbol{V}}(\varphi)$. We can thus interpret a conjunction of affine equalities as the affine hull $\mathsf{aff}(G)$ of the set $G \in \wp(\mathbb{Q}^n)$ of models of φ. We define a cube imposed by the ranges of $v_1, \ldots v_n$.

Definition 5.2. *Let $v_{1,\ell}, v_{1,u}, \ldots, v_{n,\ell}, v_{n,u} \in \mathbb{Q}$ such that $v_{i,\ell} \leq v_i \leq v_{i,u}$ for all $1 \leq i \leq n$. Define the cube $C \in \wp(\mathbb{Q}^n)$ that encloses $(v_1, \ldots, v_n)$ as:*

$$C \;=\; \{(x_1, \ldots, x_n) \in \mathbb{Q}^n \mid \forall 1 \leq i \leq n : v_{i,\ell} \leq x_i \leq v_{i,u}\}$$

Corollary 5.1. *Let C be defined as in Def. 5.2 and let $G \in \mathbb{Q}^n$ be an affine basis with $g \models \varphi$ for all $g \in G$. Then:*

$$\begin{array}{rcl} C \cap \mathit{aff}(G) & = & \{(x_1, \ldots, x_n) \in \mathit{aff}(G) : (x_1, \ldots, x_n) \in C\} \\ & = & \{(x_1, \ldots, x_n) \in \mathit{aff}(G) : \forall 1 \leq i \leq n : v_{i,\ell} \leq x_i \leq v_{i,u}\} \end{array}$$

Let $(x_1, \ldots, x_n) \in C \cap \mathsf{aff}(G)$ and assume, without loss of generality, that x_1 is maximal. Hence, there exist $\lambda_2, \ldots, \lambda_n, c \in \mathbb{Q}$ such that $x_1 = c + \sum_{i=2}^{n} \lambda_i \cdot x_i$ is maximal. Clearly, maximality of x_1 entails maximality of $\sum_{i=2}^{n} \lambda_i \cdot x_i$, which is maximal iff the summands $\lambda_i \cdot x_i$ are maximal. Further, we have the assumption $v_{i,\ell} \leq x_i \leq v_{i,u}$ for all $2 \leq i \leq n$. We perform a case distinction on the sign of λ_i:

- $\lambda_i < 0$ whence $\lambda_i \cdot v_{i,\ell} = \max\{\lambda_i \cdot x \mid v_{i,\ell} \leq x \leq v_{i,u}\}$;
- $\lambda_i \geq 0$ whence $\lambda_i \cdot v_{i,u} = \max\{\lambda_i \cdot x \mid v_{i,\ell} \leq x \leq v_{i,u}\}$.

With linearity of addition over $\mathbb{Q}$ and analogous treatment of minimization, we obtain a correctness argument for Prop. 5.1.

5.1.2 Lifting Affine Equalities to Octagons

Consider now the more general problem of deriving transfer functions for octagons by lifting affine equalities.[1] As an example, consider a formula φ that encodes the assembly code fragment `ADD R0 R1; LSL R0`, where `ADD R0 R1` sums up the values of `R0` and `R1` and stores the result in `R0`, whereas `LSL R0` shifts `R0` one bit to the left. Suppose that neither `ADD` nor `LSL` over- or underflows. Computing the affine abstraction of φ over bit-vectors $\boldsymbol{r0}$, $\boldsymbol{r0}$, $\boldsymbol{r0'}$, and $\boldsymbol{r1'}$ yields two equalities:

$$\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle \;=\; 2 \cdot \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + 2 \cdot \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \qquad \langle\!\langle \boldsymbol{r1'} \rangle\!\rangle \;=\; \langle\!\langle \boldsymbol{r1} \rangle\!\rangle$$

We aim to construct an update that maps octagonal inputs (with symbolic constants) to octagonal outputs (likewise with symbolic constants). Hence, we aim to compute the following map from $\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = 2 \cdot \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + 2 \cdot \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \wedge \langle\!\langle \boldsymbol{r1'} \rangle\!\rangle = \langle\!\langle \boldsymbol{r1} \rangle\!\rangle$:

$$\left\{\begin{array}{rcl} \langle\!\langle \boldsymbol{r0} \rangle\!\rangle & \leq & d_1 \\ \langle\!\langle \boldsymbol{r1} \rangle\!\rangle & \leq & d_2 \\ -\langle\!\langle \boldsymbol{r0} \rangle\!\rangle & \leq & d_3 \\ -\langle\!\langle \boldsymbol{r1} \rangle\!\rangle & \leq & d_4 \\ \hline \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle & \leq & d_5 \\ -\langle\!\langle \boldsymbol{r0} \rangle\!\rangle - \langle\!\langle \boldsymbol{r1} \rangle\!\rangle & \leq & d_6 \\ -\langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle & \leq & d_7 \\ \langle\!\langle \boldsymbol{r0} \rangle\!\rangle - \langle\!\langle \boldsymbol{r1} \rangle\!\rangle & \leq & d_8 d \end{array}\right\} \rightsquigarrow \left\{\begin{array}{rcl} \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle & \leq & 2 \cdot d_1 + 2 \cdot d_2 \\ \langle\!\langle \boldsymbol{r1'} \rangle\!\rangle & \leq & d_2 \\ -\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle & \leq & 2 \cdot d_3 + 2 \cdot d_4 \\ -\langle\!\langle \boldsymbol{r1'} \rangle\!\rangle & \leq & d_4 \\ \hline \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1'} \rangle\!\rangle & \leq & 2 \cdot d_1 + 3 \cdot d_2 \\ -\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle - \langle\!\langle \boldsymbol{r1'} \rangle\!\rangle & \leq & 2 \cdot d_3 + 4 \cdot d_4 \\ -\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1'} \rangle\!\rangle & \leq & 2 \cdot (d_3 + d_4) + d_2 \\ \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle - \langle\!\langle \boldsymbol{r1'} \rangle\!\rangle & \leq & 2 \cdot (d_1 + d_2) + d_4 \end{array}\right\}$$

We do so by constructing an update operation that uses the unary input constraints only (those which appear above the bar separator on the left). We modify the lifting procedure from the previous section so as to express output constraints in terms of symbolic constants $d_1, \ldots, d_8$ from the input constraints. Analogously to before, we obtain the four unary output constraints on the right, simply by substituting symbolic minimal and maximal values. For example, from $-d_3 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq d_1$, $-d_4 \leq \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq d_2$, and $\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = 2 \cdot \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle$, we deduce $-2 \cdot d_3 - 2 \cdot d_4 \leq \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle \leq 2 \cdot d_1 + 2 \cdot d_2$ using substitution (corresponding to rows 1 and 3 in the above output octagon). The binary output constraints are derived using linear

[1] One might be forgiven for thinking that the easiest way to apply an affine transformer to an octagon is to compute an affine transformation of the constraints. Unfortunately, the output of such an approach is in general a convex polyhedron rather than an octagon. The standard technique for describing a convex polyhedron by an octagon is complicated and costly in itself: (1) convert the polyhedron into a frame representation, (2) enumerate its vertices $p_1, \ldots, p_k \in \mathbb{Z}^n$, and (3) compute the join $\bigsqcup_{i=1}^{k} \alpha_{\mathsf{oct}}(p_k)$. More details are given in [166, Sect. 4.3]. Of course, (integer) linear programming can also be applied to this task [3].

combinations of the unary output constraints. For instance, the symbolic upper bound of a constraint $-\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1'} \rangle\!\rangle$ is computed from the unary constraints $-\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle \leq 2 \cdot d_3 + 2 \cdot d_4$ and $\langle\!\langle \boldsymbol{r1'} \rangle\!\rangle \leq d_2$ by substituting the right-hand sides of the inequalities:

$$-\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1'} \rangle\!\rangle \quad \leq \quad (2 \cdot d_3 + 2 \cdot d_4) + (d_2)$$

Likewise, the other binary output constraints are obtained. It is important to note that, since the output constraints do no use relational information from the inputs (such as $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 5$), we obtain a sub-optimal update.

Example 5.3. *Suppose an input octagon that describes* $0 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq 4$, $0 \leq \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 1$, *and* $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 4$. *We then derive:*

$$0 \leq \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle \leq 10 \qquad 0 \leq \langle\!\langle \boldsymbol{r1'} \rangle\!\rangle \leq 1 \qquad 0 \leq \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1'} \rangle\!\rangle \leq 11$$

An optimal transfer function, however, would derive:

$$0 \leq \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle \leq 8 \qquad 0 \leq \langle\!\langle \boldsymbol{r1'} \rangle\!\rangle \leq 1 \qquad 0 \leq \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1'} \rangle\!\rangle \leq 8$$

Although the above method fails to propagate the effects of some inputs into the output constraints, it retains the attractive property that a sound octagonal update can be constructed straightforwardly by lifting the affine relations. Interestingly, Miné [166, Fig. 27] also discusses the relative precision of transformers for octagons, though in his discussion the base semantics is polyhedral rather than Boolean. Using his classification, precision of the transfer functions derived using affine abstraction followed by lifting could be described as *medium*.

5.1.3 Lifting Polynomial Equalities to Intervals

To illustrate the process of lifting polynomial updates to intervals, consider again the equality $\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = \langle\!\langle \boldsymbol{r1} \rangle\!\rangle + \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2} \rangle\!\rangle$, the computation of which we have described for the assembly fragment `MUL R0 R2; ADD R0 R1` in Chap. 4.2.6. Of course, this polynomial neither relates internal bounds nor symbolic constants on input or output intervals. One could expect that non-linear equalities can be straightforwardly lifted to range updates using the techniques from Chap. 5.1.1 or Chap. 5.1.2. However, this is not the case.

Example 5.4. *Suppose we lift the polynomial* $\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = \langle\!\langle \boldsymbol{r1} \rangle\!\rangle + \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2} \rangle\!\rangle$ *to intervals as before. This operation gives:*

$$\langle\!\langle \boldsymbol{r0'_\ell} \rangle\!\rangle = \langle\!\langle \boldsymbol{r1_\ell} \rangle\!\rangle + \langle\!\langle \boldsymbol{r0_\ell} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2_\ell} \rangle\!\rangle \qquad \langle\!\langle \boldsymbol{r0'_u} \rangle\!\rangle = \langle\!\langle \boldsymbol{r1_u} \rangle\!\rangle + \langle\!\langle \boldsymbol{r0_u} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2_u} \rangle\!\rangle$$

Further, assume inputs

$$\begin{array}{lll} \langle\!\langle \boldsymbol{r0_\ell} \rangle\!\rangle = -4 & \langle\!\langle \boldsymbol{r1_\ell} \rangle\!\rangle = 0 & \langle\!\langle \boldsymbol{r2_\ell} \rangle\!\rangle = 2 \\ \langle\!\langle \boldsymbol{r0_u} \rangle\!\rangle = 2 & \langle\!\langle \boldsymbol{r1_u} \rangle\!\rangle = 2 & \langle\!\langle \boldsymbol{r2_u} \rangle\!\rangle = 2 \end{array}$$

which define $\langle\!\langle \boldsymbol{r0'_\ell} \rangle\!\rangle = 8$ *and* $\langle\!\langle \boldsymbol{r0'_u} \rangle\!\rangle = 6$, *a result that is obviously incorrect.*

Lifting polynomials by applying the technique discussed for affine equalities gives incorrect results because monotonic transformers on $\mathbb{Q}$ are assumed, allowing to syntactically extract the extremal values from input ranges. However, polynomials are non-monotonic. We circumvent this problem by augmenting the lifted terms with minimization and maximization operators as in the following example.

Example 5.5. *Consider again the polynomial* $\langle\!\langle \boldsymbol{r0}' \rangle\!\rangle = \langle\!\langle \boldsymbol{r1} \rangle\!\rangle + \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2} \rangle\!\rangle$ *as in Ex. 5.4. Since the term* $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2} \rangle\!\rangle$ *is non-monotonic, it is not possible to determine which of the monomials* $\langle\!\langle \boldsymbol{r0}_\ell \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2}_\ell \rangle\!\rangle$, $\langle\!\langle \boldsymbol{r0}_\ell \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2}_u \rangle\!\rangle$, $\langle\!\langle \boldsymbol{r0}_u \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2}_\ell \rangle\!\rangle$, *and* $\langle\!\langle \boldsymbol{r0}_u \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2}_u \rangle\!\rangle$ *yields the least (resp. greatest) value. Further,* 0 *may be an extremal value not determined through the bounds of* $\boldsymbol{r0}$ *and* $\boldsymbol{r2}$*. We thus evaluate monomials at runtime, during application of a transfer function, which gives:*

$$\begin{array}{rcl} \langle\!\langle \boldsymbol{r0}'_\ell \rangle\!\rangle & = & \langle\!\langle \boldsymbol{r1}_\ell \rangle\!\rangle + \min \left\{ \begin{array}{l} \langle\!\langle \boldsymbol{r0}_\ell \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2}_\ell \rangle\!\rangle, \langle\!\langle \boldsymbol{r0}_\ell \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2}_u \rangle\!\rangle, \\ \langle\!\langle \boldsymbol{r0}_u \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2}_\ell \rangle\!\rangle, \langle\!\langle \boldsymbol{r0}_u \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2}_u \rangle\!\rangle, 0 \end{array} \right\} \\ \langle\!\langle \boldsymbol{r0}'_u \rangle\!\rangle & = & \langle\!\langle \boldsymbol{r1}_u \rangle\!\rangle + \max \left\{ \begin{array}{l} \langle\!\langle \boldsymbol{r0}_\ell \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2}_\ell \rangle\!\rangle, \langle\!\langle \boldsymbol{r0}_\ell \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2}_u \rangle\!\rangle, \\ \langle\!\langle \boldsymbol{r0}_u \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2}_\ell \rangle\!\rangle, \langle\!\langle \boldsymbol{r0}_u \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2}_u \rangle\!\rangle, 0 \end{array} \right\} \end{array}$$

Assume the same input ranges as in Ex. 5.4. Then, we obtain the correct result:

$$\begin{array}{rcl} \langle\!\langle \boldsymbol{r0}'_\ell \rangle\!\rangle & = & 0 + \min\{8, -8, -4, 4, 0\} \quad = \quad -8 \\ \langle\!\langle \boldsymbol{r0}'_u \rangle\!\rangle & = & 2 + \max\{8, -8, -4, 4, 0\} \quad = \quad 8 \end{array}$$

This form of lifted update for polynomials involves, respectively, minimization and maximization operations. These operations are required because it is not until the symbolic variables are instantiated that the relative sizes of the non-linear terms can be compared; these comparisons are redundant for linear terms due to monotonicity. To present this transformation formally, let $\boldsymbol{V}_{\mathsf{in}}$ and $\boldsymbol{V}_{\mathsf{out}}$ denote the input and output variables as before. Further, denote by $\mathcal{S}$ a set of template monomials over the bit-vectors $\boldsymbol{V} = \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\}$. Thus, if $s \in \mathcal{S}$, then $s = \Pi_{i=1}^n \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle^{e_{i,s}}$ with constants $e_{1,s}, \ldots, e_{n,s} \in \mathbb{N}$. Formally, a polynomial update is thus represented as:

$$\begin{array}{rcl} \langle\!\langle \boldsymbol{v}' \rangle\!\rangle & = & \sum_{i=1}^n \lambda_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle + \sum_{s \in \mathcal{S}} \lambda_s \cdot s + c \\ & = & \sum_{i=1}^n \lambda_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle + \sum_{s \in \mathcal{S}} \lambda_s \cdot \Pi_{i=1}^n \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle^{e_{i,s}} + c \end{array}$$

Definition 5.3. *We define a map* $\mu(s)$ *to generate the set of all permutations of* $s = \Pi_{i=1}^n \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle^{e_{i,s}}$ *with bit-vectors* $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n$ *lifted to symbolic range constraints:*

$$\mu(s) \quad = \quad \{\Pi_{i=1}^n \langle\!\langle \boldsymbol{z}_i \rangle\!\rangle^{e_i} \mid i \in \{1, \ldots, n\} \wedge \boldsymbol{z}_i \in \{\boldsymbol{v}_{i,\ell}, \boldsymbol{v}_{i,u}\}\}$$

Definition 5.4. *Let* $\lambda \in \mathbb{Z}$ *and* $s = \Pi_{i=1}^n \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle^{e_{i,s}}$ *with* $e_{1,s}, \ldots, e_{n,s} \in \mathbb{N}$*. We define a map* $\sigma(\lambda, s)$ *to extract either the minimum or the maximum value of* s *from* $\mu(s)$*, depending on the sign of* λ*:*

$$\sigma(\lambda, s) \quad = \quad \begin{cases} \min(\mu(s) \cup Z) & : \textit{ if } \lambda < 0 \\ \max(\mu(s) \cup Z) & : \textit{ otherwise} \end{cases}$$

Here, $Z \in \{\emptyset, \{0\}\}$ *is defined:*

$$Z \quad = \quad \begin{cases} \{0\} & : \exists i \in \{1, \ldots, n\} : 0 \in [\langle\!\langle \boldsymbol{v}_{i,\ell} \rangle\!\rangle, \langle\!\langle \boldsymbol{v}_{i,u} \rangle\!\rangle] \wedge e^{i,s} \neq 0 \\ \emptyset & : \textit{ otherwise} \end{cases}$$

For example, if $s = \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2} \rangle\!\rangle$, then:

$$\mu(s) \quad = \quad \{\langle\!\langle \boldsymbol{r0}_\ell \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2}_\ell \rangle\!\rangle, \langle\!\langle \boldsymbol{r0}_\ell \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2}_u \rangle\!\rangle, \langle\!\langle \boldsymbol{r0}_u \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2}_\ell \rangle\!\rangle, \langle\!\langle \boldsymbol{r0}_u \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2}_u \rangle\!\rangle\}$$

Observe that considering 0 as an extremal value may be necessary.

Example 5.6. *Let* $\langle\!\langle \boldsymbol{v}' \rangle\!\rangle = \langle\!\langle \boldsymbol{v} \rangle\!\rangle^2$ *define an update and suppose inputs* $\langle\!\langle \boldsymbol{v}_\ell \rangle\!\rangle = -2$ *and* $\langle\!\langle \boldsymbol{v}_u \rangle\!\rangle = 2$*. Then,* $\langle\!\langle \boldsymbol{v}_\ell \rangle\!\rangle^2 = 4$ *and* $\langle\!\langle \boldsymbol{v}_u \rangle\!\rangle^2 = 4$*, yet* $\langle\!\langle \boldsymbol{v}' \rangle\!\rangle \in [0, 4]$*.*

On the other hand, always adding 0 as an extremal candidate may be overly pessimistic, which can be seen for $\langle\!\langle \boldsymbol{v}' \rangle\!\rangle = \langle\!\langle \boldsymbol{v} \rangle\!\rangle^2$ with inputs $\langle\!\langle \boldsymbol{v}_\ell \rangle\!\rangle = 2$ and $\langle\!\langle \boldsymbol{v}_u \rangle\!\rangle = 4$. Optimal output boundaries are then defined as $\langle\!\langle \boldsymbol{v}'_\ell \rangle\!\rangle = 4$ and $\langle\!\langle \boldsymbol{v}'_u \rangle\!\rangle = 16$. Hence, we consider 0 as a candidate upon evaluation of a transformer iff $\langle\!\langle \boldsymbol{v}_{i,\ell} \rangle\!\rangle \leq 0 \leq \langle\!\langle \boldsymbol{v}_{i,u} \rangle\!\rangle$ for some $\boldsymbol{v}_i$ that appears in the monomial with a non-zero exponent. This is because a monomial $s = \Pi_{i=1}^n \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle^{e_{i,s}}$ has roots for $\langle\!\langle \boldsymbol{v}_i \rangle\!\rangle = 0$ only. Without loss of generality, each polynomially extended equation takes the form

$$\langle\!\langle \boldsymbol{v}' \rangle\!\rangle \quad = \quad \textstyle\sum_{i=1}^n \lambda_i \cdot \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle + \sum_{s \in \mathcal{S}} \lambda_s \cdot s + c$$

where $\boldsymbol{v}' \in \boldsymbol{V}'$, $\lambda_1, \ldots, \lambda_n \in \mathbb{Q}$, and $\lambda_s \in \mathbb{Q}$ for all $s \in \mathcal{S}$. With β defined as in Chap. 5.1.1, we then replace each polynomial by a pair of equations as follows:

$$\begin{array}{lcllll} \langle\!\langle \boldsymbol{v}'_\ell \rangle\!\rangle & = & \sum_{i=1}^n \lambda_i \cdot \beta(-\lambda_i, \boldsymbol{v}_i) & + & \sum_{s \in \mathcal{S}} \lambda_s \cdot \sigma(-\lambda_s, s) & + \quad c \\ \langle\!\langle \boldsymbol{v}'_u \rangle\!\rangle & = & \sum_{i=1}^n \lambda_i \cdot \beta(\lambda_i, \boldsymbol{v}_i) & + & \sum_{s \in \mathcal{S}} \lambda_s \cdot \sigma(\lambda_s, s) & + \quad c \end{array}$$

Note that linear terms are transformed as before: it is only non-linear monomials that require special treatment.

Proposition 5.2. *Let* $s = \Pi_{i=1}^n \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle^{e_i}$ *with* $e_1, \ldots, e_n \in \mathbb{N}$*. Then,* s *constrained so that either* $\langle\!\langle \boldsymbol{v}_i \rangle\!\rangle \leq 0$ *or* $0 \leq \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle$ *for all* $1 \leq i \leq n$ *defines a monotone function.*

Proof. Correctness follows directly from monotonicity of multiplication over $\mathbb{N}$. □

The following proposition shows that lifting a monomial s to minimal and maximal values of symbolic ranges by applying $\gamma(\lambda, s)$ indeed yields the extremal values.

Proposition 5.3. *Put* $\boldsymbol{V} = \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\}$ *and let* $\boldsymbol{v}_{1,\ell}, \boldsymbol{v}_{1,u}, \ldots, \boldsymbol{v}_{n,\ell}, \boldsymbol{v}_{n,u}$ *such that* $\langle\!\langle \boldsymbol{v}_{i,\ell} \rangle\!\rangle \leq \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle \leq \langle\!\langle \boldsymbol{v}_{i,u} \rangle\!\rangle$ *for all* $\boldsymbol{v}_i \in \boldsymbol{V}$*. Further, let* $s = \Pi_{i=1}^n \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle^{e_i}$ *denote a monomial over* $\boldsymbol{V}$*. If* $0 \notin [\langle\!\langle \boldsymbol{v}_{i,\ell} \rangle\!\rangle, \langle\!\langle \boldsymbol{v}_{i,u} \rangle\!\rangle]$ *for all* $1 \leq i \leq n$*, then:*

$$\begin{array}{rcl} \min(\mu(s)) & = & \min\{x \in \mathbb{Z} \mid x_i \in \mathbb{Z} : \langle\!\langle \boldsymbol{v}_{i,\ell} \rangle\!\rangle \leq x_i \leq \langle\!\langle \boldsymbol{v}_{i,u} \rangle\!\rangle \wedge x = \Pi_{i=1}^n x_i^{e_i}\} \\ \max(\mu(s)) & = & \max\{x \in \mathbb{Z} \mid x_i \in \mathbb{Z} : \langle\!\langle \boldsymbol{v}_{i,\ell} \rangle\!\rangle \leq x_i \leq \langle\!\langle \boldsymbol{v}_{i,u} \rangle\!\rangle \wedge x = \Pi_{i=1}^n x_i^{e_i}\} \end{array}$$

If $0 \in [\langle\!\langle \boldsymbol{v}_{i,\ell} \rangle\!\rangle, \langle\!\langle \boldsymbol{v}_{i,u} \rangle\!\rangle]$*, then:*

$$\begin{array}{rcl} \min(\mu(s) \cup \{0\}) & = & \min\{x \in \mathbb{Z} \mid x_i \in \mathbb{Z} : \langle\!\langle \boldsymbol{v}_{i,\ell} \rangle\!\rangle \leq x_i \leq \langle\!\langle \boldsymbol{v}_{i,u} \rangle\!\rangle \wedge x = \Pi_{i=1}^n x_i^{e_i}\} \\ \max(\mu(s) \cup \{0\}) & = & \max\{x \in \mathbb{Z} \mid x_i \in \mathbb{Z} : \langle\!\langle \boldsymbol{v}_{i,\ell} \rangle\!\rangle \leq x_i \leq \langle\!\langle \boldsymbol{v}_{i,u} \rangle\!\rangle \wedge x = \Pi_{i=1}^n x_i^{e_i}\} \end{array}$$

Proof. We sketch a proof by induction for the latter case, which is more general than the first one. For the base clause, we observe that $s_1 = \langle\!\langle \boldsymbol{v}_1 \rangle\!\rangle^{e_1}$ has a single root, namely $\langle\!\langle \boldsymbol{v}_1 \rangle\!\rangle = 0$. With monotonicity of multiplication, we deduce

$$\begin{array}{rl} & \min\{x \in \mathbb{Z} \mid x \in \mathbb{Z} : \langle\!\langle \boldsymbol{v}_{1,\ell} \rangle\!\rangle \leq x_1 \leq \langle\!\langle \boldsymbol{v}_{1,u} \rangle\!\rangle \wedge x = x_1^{e_1}\} \\ = & \min\{x \in \mathbb{Z} \mid x \in \mathbb{Z} : (x_1 = \langle\!\langle \boldsymbol{v}_{1,\ell} \rangle\!\rangle = x_1 \vee \langle\!\langle \boldsymbol{v}_{1,u} \rangle\!\rangle = x_1 \vee x_1 = 0) \wedge x = x_1^{e_1}\} \\ = & \min\{\langle\!\langle \boldsymbol{v}_{1,\ell} \rangle\!\rangle, \langle\!\langle \boldsymbol{v}_{1,u} \rangle\!\rangle, 0\} \\ = & \min(\mu(s_1) \cup \{0\}) \end{array}$$

as desired. The proof for $\max(\mu(s_1))$ is similar. Assume the proposition holds for $s_n = \Pi_{i=1}^n \langle\!\langle \boldsymbol{v}_i \rangle\!\rangle^{e_i}$ and let $s_{n+1} = s_n \cdot \langle\!\langle \boldsymbol{v}_{n+1} \rangle\!\rangle^{e_{n+1}}$. Then:

$$\begin{array}{rl} & \min\{x \in \mathbb{Z} \mid x_i \in \mathbb{Z} : \langle\!\langle \boldsymbol{v}_{i,\ell} \rangle\!\rangle \leq x_i \leq \langle\!\langle \boldsymbol{v}_{i,u} \rangle\!\rangle \wedge x = \Pi_{i=1}^{n+1} x_i^{e_i}\} \\ = & \min\{x \in \mathbb{Z} \mid x_i \in \mathbb{Z} : \langle\!\langle \boldsymbol{v}_{i,\ell} \rangle\!\rangle \leq x_i \leq \langle\!\langle \boldsymbol{v}_{i,u} \rangle\!\rangle \wedge x = \Pi_{i=1}^n x_i^{e_i} \cdot x_{n+1}^{e_{n+1}}\} \\ = & \min\left\{ x \in \mathbb{Z} \,\middle|\, \begin{array}{l} x_i \in \mathbb{Z} : \langle\!\langle \boldsymbol{v}_{i,\ell} \rangle\!\rangle \leq x_i \leq \langle\!\langle \boldsymbol{v}_{i,u} \rangle\!\rangle \wedge \\ (x = \Pi_{i=1}^n x_i^{e_i} \cdot x_{n+1,\ell} \vee x = \Pi_{i=1}^n x_i^{e_i} \cdot x_{n+1,u} \vee x = 0) \end{array} \right\} \\ = & \min\left\{ \begin{array}{l} x \cdot \langle\!\langle \boldsymbol{v}_{n+1,\ell} \rangle\!\rangle^{e_{n+1}}, \\ x \cdot \langle\!\langle \boldsymbol{v}_{n+1,u} \rangle\!\rangle^{e_{n+1}}, \\ 0 \end{array} \,\middle|\, x_i \in \mathbb{Z} : \langle\!\langle \boldsymbol{v}_{i,\ell} \rangle\!\rangle \leq x_i \leq \langle\!\langle \boldsymbol{v}_{i,u} \rangle\!\rangle \wedge x = \Pi_{i=1}^n x_i^{e_i} \right\} \\ = & \min\left(\{m \cdot \langle\!\langle \boldsymbol{v}_{n+1,\ell} \rangle\!\rangle^{e_{n+1}}, m \cdot \langle\!\langle \boldsymbol{v}_{n+1,u} \rangle\!\rangle^{e_{n+1}} \mid m \in \mu(s_n)\} \cup \{0\}\right) \\ = & \min\left(\{m \mid m \in \mu(s_{n+1})\} \cup \{0\}\right) \\ = & \min(\mu(s_{n+1}) \cup \{0\}) \end{array}$$

Correctness of the remaining cases can be shown accordingly. □

5.1.4 Lifting Polynomial Equalities to Octagons

Similar in spirit to the technique introduced to lift polynomial updates to intervals, we derive polynomial updates for octagons. From a polynomial update

$$\begin{array}{rcl} \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle & = & 2 \cdot \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + 2 \cdot \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \wedge \langle\!\langle \boldsymbol{r1'} \rangle\!\rangle = \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \\ \langle\!\langle \boldsymbol{r1'} \rangle\!\rangle & = & \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \\ \langle\!\langle \boldsymbol{r2'} \rangle\!\rangle & = & \langle\!\langle \boldsymbol{r2} \rangle\!\rangle \end{array}$$

that relates variables, we thus aim to construct an update for octagons.[2] The strategy applied to do so is a combination of the techniques from Chap. 5.1.2 and Chap. 5.1.3. Again, we do so by using the unary constraints $d_1, \ldots, d_6$ of an input octagon (those constraints found above the horizontal bar)

$$\left\{\begin{array}{r|r|r} \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq d_1 & \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq d_2 & \langle\!\langle \boldsymbol{r2} \rangle\!\rangle \leq d_3 \\ -\langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq d_4 & -\langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq d_5 & -\langle\!\langle \boldsymbol{r2} \rangle\!\rangle \leq d_6 \\ \hline \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq d_7 & \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r2} \rangle\!\rangle \leq d_{11} & \langle\!\langle \boldsymbol{r1} \rangle\!\rangle + \langle\!\langle \boldsymbol{r2} \rangle\!\rangle \leq d_{15} \\ -\langle\!\langle \boldsymbol{r0} \rangle\!\rangle - \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq d_8 & -\langle\!\langle \boldsymbol{r0} \rangle\!\rangle - \langle\!\langle \boldsymbol{r2} \rangle\!\rangle \leq d_{12} & -\langle\!\langle \boldsymbol{r1} \rangle\!\rangle - \langle\!\langle \boldsymbol{r2} \rangle\!\rangle \leq d_{16} \\ -\langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq d_9 & -\langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r2} \rangle\!\rangle \leq d_{13} & -\langle\!\langle \boldsymbol{r1} \rangle\!\rangle + \langle\!\langle \boldsymbol{r2} \rangle\!\rangle \leq d_{17} \\ \langle\!\langle \boldsymbol{r0} \rangle\!\rangle - \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq d_{10} & \langle\!\langle \boldsymbol{r0} \rangle\!\rangle - \langle\!\langle \boldsymbol{r2} \rangle\!\rangle \leq d_{14} & \langle\!\langle \boldsymbol{r1} \rangle\!\rangle - \langle\!\langle \boldsymbol{r2} \rangle\!\rangle \leq d_{18} \end{array}\right\}$$

that characterizes $\boldsymbol{r0}$, $\boldsymbol{r1}$ and $\boldsymbol{r2}$, to describe symbolic constants $d'_1, \ldots, d'_{18}$ of an output octagon over $\boldsymbol{r0'}$, $\boldsymbol{r1'}$, and $\boldsymbol{r2'}$. In the output (with the respective primed variables), we substitute the polynomial update into the left-hand side of each constraint. For instance, the output constraint $\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle - \langle\!\langle \boldsymbol{r1'} \rangle\!\rangle$ is transformed into:

$$(\langle\!\langle \boldsymbol{r1} \rangle\!\rangle + \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2} \rangle\!\rangle) - \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \quad \leq \quad d'_{10}$$

First of all, we simplify $(\langle\!\langle \boldsymbol{r1} \rangle\!\rangle + \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2} \rangle\!\rangle) - \langle\!\langle \boldsymbol{r1} \rangle\!\rangle$, which reduces the inequality to $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2} \rangle\!\rangle \leq d'_{10}$. To compute d'_{10}, we handle linear and non-linear terms separately. Linear terms are treated as in Chap. 5.1.2 by substituting a variable with a positive (resp. negative) coefficient by its upper (resp. lower) bound:

$$\begin{array}{llll} \langle\!\langle \boldsymbol{r0} \rangle\!\rangle & \rightsquigarrow & d_1 & -\langle\!\langle \boldsymbol{r0} \rangle\!\rangle \quad \rightsquigarrow \quad d_4 \qquad \langle\!\langle \boldsymbol{r2} \rangle\!\rangle \quad \rightsquigarrow \quad d_3 \\ \langle\!\langle \boldsymbol{r1} \rangle\!\rangle & \rightsquigarrow & d_2 & -\langle\!\langle \boldsymbol{r1} \rangle\!\rangle \quad \rightsquigarrow \quad d_5 \qquad -\langle\!\langle \boldsymbol{r2} \rangle\!\rangle \quad \rightsquigarrow \quad d_6 \end{array}$$

For example, $\langle\!\langle \boldsymbol{r1} \rangle\!\rangle$ (resp. $-\langle\!\langle \boldsymbol{r1} \rangle\!\rangle$) is thus turned into d_2 (resp. d_5). The non-linear term $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2} \rangle\!\rangle$ (resp. $-\langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2} \rangle\!\rangle$) is handled using maximization (resp. minimization) of the bounds of $\boldsymbol{r0}$ and $\boldsymbol{r2}$ at runtime, respectively. With the equivalence $\langle\!\langle \boldsymbol{v} \rangle\!\rangle \leq d$ iff $-d \leq -\langle\!\langle \boldsymbol{v} \rangle\!\rangle$, these combinations of non-linear terms are:

$$\mathcal{S}_{\langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2} \rangle\!\rangle} \quad = \quad \{d_1 \cdot d_3, d_1 \cdot (-d_6), (-d_4) \cdot d_3, (-d_4) \cdot (-d_6)\}$$

Likewise, monomials with a negative coefficient are handled using minimization. Combining the transformed terms for linear and non-linear ones thus simply gives:

$$\begin{array}{rcl} \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle - \langle\!\langle \boldsymbol{r1'} \rangle\!\rangle & = & \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2} \rangle\!\rangle \\ & \leq & \max\{s \in \mathcal{S}_{\langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2} \rangle\!\rangle} \cup Z\} \end{array}$$

[2] Observe that Miné [166] does not address multiplication. However, it is possible to resort to convex polyhedra, where different ways of expressing multiplication were already studied by Cousot and Halbwachs [82], and then convert the result back into an octagon [166, Sect. 4.3].

Z is defined as in Def. 5.4. Likewise, we obtain:

$$\begin{array}{rcl} -\langle\!\langle \boldsymbol{r0}' \rangle\!\rangle - \langle\!\langle \boldsymbol{r1}' \rangle\!\rangle & = & -2 \cdot \langle\!\langle \boldsymbol{r1} \rangle\!\rangle - \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2} \rangle\!\rangle \\ & \leq & 2 \cdot d_5 - \min\{s \in \mathcal{S}_{\langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2} \rangle\!\rangle} \cup Z\} \end{array}$$

For brevity, define $s_\ell = \min\{s \in \mathcal{S}_{\langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r1} \rangle\!\rangle} \cup Z\}$ and $s_u = \max\{s \in \mathcal{S}_{\langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2} \rangle\!\rangle} \cup Z\}$. Applying the transformation sketched above to the overall output octagon yields:

$$\left\{ \begin{array}{lcl|lcl|lcl} d'_1 & = & d_2 + s_u & d'_2 & = & d_2 & d'_3 & = & d_3 \\ d'_4 & = & d_5 - s_\ell & d'_5 & = & d_5 & d'_6 & = & d_6 \\ \hline d'_7 & = & 2 \cdot d_2 + s_u & d'_{11} & = & d_2 + s_u + d_3 & d'_{15} & = & d_2 + d_3 \\ d'_8 & = & 2 \cdot d_5 - s_\ell & d'_{12} & = & d_5 - s_\ell + d_6 & d'_{16} & = & d_5 + d_6 \\ d'_9 & = & -s_\ell & d'_{13} & = & d_5 - s_\ell + d_3 & d'_{17} & = & d_5 + d_3 \\ d'_{10} & = & s_u & d'_{14} & = & d_2 + s_u + d_6 & d'_{18} & = & d_2 + d_6 \end{array} \right\}$$

Example 5.7. *Suppose the block* `MUL R0 R2; ADD R0 R1` *is entered with:*

$$\left\{ \begin{array}{c|c|c} d_1 = 10 & d_2 = 5 & d_3 = 10 \\ d_4 = -1 & d_5 = -5 & d_6 = -5 \end{array} \right\}$$

The remaining constraints are omitted as they are not used to specify the polynomial update for octagons when lifting is applied. The combinations of the monomial $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2} \rangle\!\rangle$ *over the symbolic bounds are then given as* $\mathcal{S}_{\langle\!\langle \boldsymbol{r0} \rangle\!\rangle \cdot \langle\!\langle \boldsymbol{r2} \rangle\!\rangle} = \{100, -50, -10, 5, 0\}$. *Hence,* $s_\ell = -50$ *and* $s_u = 100$. *Applying the octagonal transformer obtained using lifting of polynomial updates, we compute:*

$$\left\{ \begin{array}{lcl|lcl|lcl} d'_1 & = & 105 & d'_2 & = & 5 & d'_3 & = & 10 \\ d'_4 & = & -5 & d'_5 & = & -5 & d'_6 & = & -5 \\ \hline d'_7 & = & 110 & d'_{11} & = & 115 & d'_{15} & = & 15 \\ d'_8 & = & 40 & d'_{12} & = & 100 & d'_{16} & = & -10 \\ d'_9 & = & 50 & d'_{13} & = & 55 & d'_{17} & = & 5 \\ d'_{10} & = & 100 & d'_{14} & = & 100 & d'_{18} & = & 0 \end{array} \right\}$$

5.2 Characterizing Linear Templates using Quantification

A formula $\varphi \in \wp(\wp(\boldsymbol{V}))$ derived from a block describes an (indirect) relation between input bit-vectors $\boldsymbol{V}_{\mathsf{in}} \subseteq \boldsymbol{V}$ and output bit-vectors $\boldsymbol{V}_{\mathsf{out}} \subseteq \boldsymbol{V}$, but not between their ranges. The relationship between concrete variables and their respective bounds, however, can be specified by augmenting φ with additional terms and quantifiers. We first sketch the mechanisms for doing so for the simple case of intervals and then generalize this approach to the more general class of template domains.

5.2.1 Specifying Optimal Intervals using Quantifiers

We study the quantifier-based approach to automatic abstraction by means of an example, namely the instruction `INC R0` in regular mode, which increments the value of register `R0` by one and stores the result in `R0`. To specify the relation between the input bit-vector $\boldsymbol{r0}$ and the outputs $\boldsymbol{r0'}$ using intervals, we introduce fresh bit-vectors $\boldsymbol{r0}_\ell$, $\boldsymbol{r0}_u$, $\boldsymbol{r0}'_\ell$, and $\boldsymbol{r0}'_u$ to represent greatest lower and least upper bounds of $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle$ and $\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle$, respectively. Further, we define symbolic expressions

$$\sigma \;=\; \langle\!\langle \boldsymbol{r0}_\ell \rangle\!\rangle \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq \langle\!\langle \boldsymbol{r0}_u \rangle\!\rangle \qquad \sigma' \;=\; \langle\!\langle \boldsymbol{r0}'_\ell \rangle\!\rangle \leq \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle \leq \langle\!\langle \boldsymbol{r0}'_u \rangle\!\rangle$$

to express that all input and output bit-vectors are confined to their respective ranges. With all bit-vectors in range, specifying the greatest lower bound $\boldsymbol{r0}'_\ell$ and least upper bound $\boldsymbol{r0}'_u \in \boldsymbol{V}^u_{\mathsf{out}}$ of $\boldsymbol{r0}$ amounts to requiring that:

1. The bit-vectors $\boldsymbol{r0}'_\ell$ and $\boldsymbol{r0}'_u$ represent respectively lower and upper bounds on the range of $\boldsymbol{r0'}$.
2. Any other lower and upper bounds on $\boldsymbol{r0'}$ are respectively less or equal to and greater or equal to $\boldsymbol{r0}'_\ell$ and $\boldsymbol{r0}'_u$.

Let φ denote the symbolic encoding of `INC R0`. We enforce the first requirement with the formula $\omega = \forall \boldsymbol{r0} : \forall \boldsymbol{r0'} : (\sigma \wedge \varphi) \Rightarrow \sigma'$. Then, $(\sigma \wedge \varphi) \Rightarrow \sigma'$ is put into CNF, which introduces existentially quantified variables. These variables are eliminated using projection before those literals that involve variables from $\boldsymbol{r0}$ and $\boldsymbol{r0'}$ are simply struck out. Let ω_{simp} denote the resulting formula in CNF, which ranges over bit-vectors $\boldsymbol{r0}_\ell$, $\boldsymbol{r0}_u$, $\boldsymbol{r0}'_\ell$, and $\boldsymbol{r0}'_u$; hence $\omega_{\mathsf{simp}} \in \wp(\wp(\{\boldsymbol{r0}_\ell, \boldsymbol{r0}_u, \boldsymbol{r0}'_\ell, \boldsymbol{r0}'_u\}))$. The second requirement is enforced by introducing auxiliary variables to represent other lower and upper bounds on $\boldsymbol{r0'}$, denoted $\boldsymbol{r0}^\star_\ell$ and $\boldsymbol{r0}^\star_u$. As before, the formula

$$\sigma^\star \;=\; \langle\!\langle \boldsymbol{r0}^\star_\ell \rangle\!\rangle \leq \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle \leq \langle\!\langle \boldsymbol{r0}^\star_u \rangle\!\rangle$$

encodes the requirement that $\langle\!\langle \boldsymbol{r0}^\star_\ell \rangle\!\rangle$ and $\langle\!\langle \boldsymbol{r0}^\star_u \rangle\!\rangle$ are lower and upper bounds of $\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle$, respectively. We then stipulate $\nu = \forall \boldsymbol{r0}^\star_\ell : \forall \boldsymbol{r0}^\star_u : \forall \boldsymbol{r0} : \forall \boldsymbol{r0'} : \kappa$ with:

$$\kappa \;=\; ((\sigma \wedge \varphi) \Rightarrow \sigma^\star) \Rightarrow (\langle\!\langle \boldsymbol{r0}^\star_\ell \rangle\!\rangle \leq \langle\!\langle \boldsymbol{r0}'_\ell \rangle\!\rangle \wedge \langle\!\langle \boldsymbol{r0}'_u \rangle\!\rangle \leq \langle\!\langle \boldsymbol{r0}^\star_u \rangle\!\rangle)$$

The left-hand side of κ requires that $\langle\!\langle \boldsymbol{r0}^\star_\ell \rangle\!\rangle$ and $\langle\!\langle \boldsymbol{r0}^\star_u \rangle\!\rangle$ are indeed lower and upper bounds of $\langle\!\langle \boldsymbol{v'} \rangle\!\rangle$. The right-hand side of the implication expresses that $\langle\!\langle \boldsymbol{r0}^\star_\ell \rangle\!\rangle$ is less or equal to $\langle\!\langle \boldsymbol{v}'_\ell \rangle\!\rangle$, and likewise that $\langle\!\langle \boldsymbol{r0}^\star_u \rangle\!\rangle$ is greater or equal to $\langle\!\langle \boldsymbol{v}'_u \rangle\!\rangle$, thereby inducing optimality of $\langle\!\langle \boldsymbol{r0}'_\ell \rangle\!\rangle$ and $\langle\!\langle \boldsymbol{r0}'_u \rangle\!\rangle$ a posteriori. In combination, this means that $\langle\!\langle \boldsymbol{r0}'_\ell \rangle\!\rangle$ (resp. $\langle\!\langle \boldsymbol{r0}'_u \rangle\!\rangle$) is the least upper (resp. greatest lower) bound of $\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle$ subject to φ. Again, we put ν into CNF and eliminate existential as well as universal quantifiers, an operation that yields a simplified formula $\nu_{\mathsf{simp}} \in \wp(\wp(\{\boldsymbol{r0}_\ell, \boldsymbol{r0}_u, \boldsymbol{r0}'_\ell, \boldsymbol{r0}'_u\}))$.

Then, $\psi = \omega_{\mathsf{simp}} \wedge \nu_{\mathsf{simp}}$ describes a direct bit-level relationship between symbolic interval bounds $\boldsymbol{r0}_\ell$ and $\boldsymbol{r0}_u$ on input and the corresponding bounds $\boldsymbol{r0}'_\ell$ and $\boldsymbol{r0}'_u$ on output. The conjoined formula ψ can thus be seen as a transformer of type $\mathsf{Int} \xrightarrow{\mathsf{Bool}} \mathsf{Int}$. However, to avoid the application of a SAT solver to evaluate the transformer, ψ can be abstracted, e.g., using conjunctions of affine equalities over $\{\boldsymbol{r0}_\ell, \boldsymbol{r0}_u, \boldsymbol{r0}'_\ell, \boldsymbol{r0}'_u\}$. Then, we obtain the expected result:

$$\alpha_{\mathsf{aff}}^{\{\boldsymbol{r0}_\ell, \boldsymbol{r0}_u, \boldsymbol{r0}'_\ell, \boldsymbol{r0}'_u\}}(\psi) \quad = \quad (\langle\!\langle \boldsymbol{r0}'_\ell \rangle\!\rangle = \langle\!\langle \boldsymbol{r0}_\ell \rangle\!\rangle + 1 \wedge \langle\!\langle \boldsymbol{r0}'_u \rangle\!\rangle = \langle\!\langle \boldsymbol{r0}_u \rangle\!\rangle + 1)$$

In the following, we show how to generalize this technique to accept specifications over arbitrary but fixed classes of templates (including intervals and octagons), and also show how this quantifier-based formulation can be instantiated to derive guards.

5.2.2 Generalization

In general, we require a transformer $\varphi \in \wp(\wp(\boldsymbol{V}))$ and $\boldsymbol{V}_{\mathsf{in}}, \boldsymbol{V}_{\mathsf{out}} \subseteq \boldsymbol{V}$ such that $\boldsymbol{V}_{\mathsf{in}} \cap \boldsymbol{V}_{\mathsf{out}} = \emptyset$. To express symbolic relations on top of φ, we augment φ with two additional types of formulae:

1. Let $\sigma(\boldsymbol{W}, \boldsymbol{T})$ encode a relation between concrete bit-vectors $\boldsymbol{W} \subseteq \boldsymbol{V}_{\mathsf{in}} \cup \boldsymbol{V}_{\mathsf{out}}$ and a symbolic encoding of constraints over bit-vectors $\boldsymbol{T}$. For example, $\sigma(\{\boldsymbol{v}\}, \{\boldsymbol{v}_\ell\}) = \langle\!\langle \boldsymbol{v}_\ell \rangle\!\rangle \leq \langle\!\langle \boldsymbol{v} \rangle\!\rangle$ encodes that the signed interpretation of a bit-vector $\boldsymbol{v}$ is greater to equal to its lower bound $\langle\!\langle \boldsymbol{v}_\ell \rangle\!\rangle$ in the interval domain.

2. Let $\nu(\boldsymbol{T}, \boldsymbol{T}')$ denote a formula that holds if a given valuation of $\boldsymbol{T}'$ entails a given valuation of $\boldsymbol{T}$. For instance, $\nu(\{\boldsymbol{x}\}, \{\boldsymbol{y}\})$ defined as $\nu(\{\boldsymbol{v}'_\ell\}, \{\boldsymbol{v}'_{\ell,\star}\}) = \langle\!\langle \boldsymbol{v}'_{\ell,\star} \rangle\!\rangle \leq \langle\!\langle \boldsymbol{v}'_\ell \rangle\!\rangle$ encodes that $\langle\!\langle \boldsymbol{v}'_\ell \rangle\!\rangle \leq \langle\!\langle \boldsymbol{v}' \rangle\!\rangle$ entails $\langle\!\langle \boldsymbol{v}'_{\ell,\star} \rangle\!\rangle \leq \langle\!\langle \boldsymbol{v}' \rangle\!\rangle$.

To construct the overall quantified specification, we first introduce fresh bit-vectors $\boldsymbol{T}_{\mathsf{in}}$ and $\boldsymbol{T}_{\mathsf{out}}$ that are used to express symbolic constraints over inputs $\boldsymbol{V}_{\mathsf{in}}$ and outputs $\boldsymbol{V}_{\mathsf{out}}$, respectively. These bit-vectors are used to encode the source domain D_1 and the target domain D_2 in the resulting transformer $D_1 \xrightarrow{\mathsf{Bool}} D_2$. There is no need to require D_1 and D_2 to be identical. For intervals, e.g., we have $\boldsymbol{T}_{\mathsf{in}} = \{\boldsymbol{v}_\ell, \boldsymbol{v}_u \mid \boldsymbol{v} \in \boldsymbol{V}_{\mathsf{in}}\}$ and $\boldsymbol{T}_{\mathsf{out}} = \{\boldsymbol{v}'_\ell, \boldsymbol{v}'_u \mid \boldsymbol{v}' \in \boldsymbol{V}_{\mathsf{out}}\}$. We also introduce a fresh set $\boldsymbol{T}_{\mathsf{out},\star}$ of bit-vectors that is identical to $\boldsymbol{T}_{\mathsf{out}}$ except for naming. To avoid accidental coupling, we require all sets of bit-vectors introduced so far to be disjoint. Then, we put:

$$\begin{array}{ll} \forall \boldsymbol{V}_{\mathsf{in}} \cup \boldsymbol{V}_{\mathsf{out}} : & (\sigma(\boldsymbol{V}_{\mathsf{in}}, \boldsymbol{T}_{\mathsf{in}}) \wedge \varphi) \Rightarrow \sigma(\boldsymbol{V}_{\mathsf{out}}, \boldsymbol{T}_{\mathsf{out}}) \quad \wedge \\ \forall \boldsymbol{V}_{\mathsf{in}} \cup \boldsymbol{V}_{\mathsf{out}} : \forall \boldsymbol{T}_{\mathsf{out},\star} : & (\sigma(\boldsymbol{V}_{\mathsf{in}}, \boldsymbol{T}_{\mathsf{in}}) \wedge \varphi) \Rightarrow \sigma(\boldsymbol{V}_{\mathsf{out}}, \boldsymbol{T}_{\mathsf{out},\star})) \Rightarrow \nu(\boldsymbol{T}_{\mathsf{out}}, \boldsymbol{T}_{\mathsf{out},\star}) \end{array}$$

Eliminating quantifiers from this formula gives a logical formula ψ drawn from $\wp(\wp(\boldsymbol{T}_{\mathsf{in}} \cup \boldsymbol{T}_{\mathsf{out}}))$. The resulting quantifier-free formula ψ can thus be abstracted

using the appropriate abstraction procedure from Chap. 4. For example, $\alpha_{\mathsf{aff}}^{\boldsymbol{T}_{\mathsf{in}} \cup \boldsymbol{T}_{\mathsf{out}}}(\psi)$ can be used to derive a conjunction of affine equalities that describes the relation between symbolic constraints $\boldsymbol{T}_{\mathsf{in}} \cup \boldsymbol{T}_{\mathsf{out}}$.

Deriving Guards using Quantification

Observe that the problem of deriving guards on the inputs can be specified as an instance of the above formulation by simply replacing $\sigma(\boldsymbol{V}_{\mathsf{out}}, \boldsymbol{T}_{\mathsf{out}})$ and $\sigma(\boldsymbol{V}_{\mathsf{out}}, \boldsymbol{T}_{\mathsf{out},\star})$ by true, respectively. Overall, this yields a simpler formula:

$$\begin{array}{lll} \forall \boldsymbol{V}_{\mathsf{in}} \cup \boldsymbol{V}_{\mathsf{out}} : & (\sigma(\boldsymbol{V}_{\mathsf{in}}, \boldsymbol{T}_{\mathsf{in}}) \wedge \varphi) & \wedge \\ \forall \boldsymbol{V}_{\mathsf{in}} \cup \boldsymbol{V}_{\mathsf{out}} : \forall \boldsymbol{T}_{\mathsf{in},\star} : & (\sigma(\boldsymbol{V}_{\mathsf{in}}, \boldsymbol{T}_{\mathsf{in},\star}) \wedge \varphi) \Rightarrow \nu(\boldsymbol{T}_{\mathsf{in}}, \boldsymbol{T}_{\mathsf{in},\star}) & \end{array}$$

Quantified variables are then eliminated from the above formula, yielding a quantifier-free formula $\psi \in \wp(\wp(\boldsymbol{T}_{\mathsf{in}}))$, which can directly be passed to solver. From the model provided for the solver, one can then directly read off the valuations for the template constraints (e.g., the upper bounds d_i for octagonal guards).

Reprise and Reflection

Quantifier-based characterizations for automatic abstraction were proposed by Monniaux [167, 169] in the setting of piecewise linear functions. Structurally, the quantifier-based construction presented here follows the same lines, with correctness arguments carrying over, too. The key difference of our technique compared to the work of Monniaux is that his base semantics consists of piecewise linear functions; he thus applies quantifier elimination directly on a piecewise linear specification of the semantics of a block. By way of contrast, we perform quantifier elimination directly on the (concrete) Boolean specification, a step which can then be followed by abstraction. As shown in Chap. 2, universal as well as quantifier elimination for Boolean formulae in CNF can easily be implemented, which contrasts with complicated and highly involved procedures for integer linear arithmetic. However, this simplicity comes at a cost:

- The complexity of quantifier elimination for linear systems typically explodes with the number of variables and constraints in a system [148], even more so if variables are discrete.

- Tractability of existential quantification for propositional Boolean formulae heavily depends on the number of Boolean variables and constraints. With increasing lengths of bit-vectors, the complexity of projection thus increases, too. Quantification-based automatic abstraction from Boolean formulae thus becomes intractable if architectures with large bit-widths are analyzed.

Table 5.1: Intermediate results for inferring exact affine transformers for octagons

	$\langle\!\langle \boldsymbol{d}'_1 \rangle\!\rangle$	$\langle\!\langle \boldsymbol{d}_1 \rangle\!\rangle$	$\langle\!\langle \boldsymbol{d}_2 \rangle\!\rangle$	$\langle\!\langle \boldsymbol{d}_3 \rangle\!\rangle$	$\langle\!\langle \boldsymbol{d}_4 \rangle\!\rangle$	$\langle\!\langle \boldsymbol{d}_5 \rangle\!\rangle$	$\langle\!\langle \boldsymbol{d}_6 \rangle\!\rangle$	$\langle\!\langle \boldsymbol{d}_7 \rangle\!\rangle$	$\langle\!\langle \boldsymbol{d}_8 \rangle\!\rangle$	$\max\langle\!\langle \boldsymbol{d}' \rangle\!\rangle$
$\mathbf{m}_1$	1	1	1	0	0	1	0	1	1	2
$\mathbf{m}_2$	8	3	3	−1	−1	5	−2	2	0	10
$\mathbf{m}_3$	22	8	7	0	1	13	3	4	0	26
$\mathbf{m}_4$	4	0	3	2	0	3	1	6	3	6

In the light of the complexity of projection for large bit-vectors, it is unsurprising that we have not been able to compute abstractions for 32-bit architectures in a reasonable amount of time for any except the smallest benchmarks. We will further elaborate on scalability in Chap. 5.5.

5.3 Interleaved Abstraction and Refinement

To derive more precise affine updates, we interleave affine abstraction with constraint inference. In this section, we discuss techniques for affine and polynomial updates on octagons as well as affine updates on arithmetical congruences.

5.3.1 Optimal Affine Updates on Octagons

To illustrate this technique by means of an example, let φ denote the propositional encoding for `ADD R0 R1; LSL R0` where again `ADD` and `LSL` operate in regular modes. Consider the inequality $\langle\!\langle \boldsymbol{r0}' \rangle\!\rangle \leq d'_1$ in the output octagon and, in particular, the problem of discovering an affine description of d'_1 using the symbolic constants $d_1, \ldots, d_8$ of the input octagon, as detailed previously.

Worked Example

We proceed by introducing sign-extended bit vectors $\boldsymbol{d}_1, \ldots, \boldsymbol{d}_8$ to represent the symbolic constants $d_1, \ldots, d_8$ of the input octagon. Further, let κ denote a Boolean formula that holds iff the eight inequalities $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq \langle\!\langle \boldsymbol{d}_1 \rangle\!\rangle, \ldots, \langle\!\langle \boldsymbol{r0} \rangle\!\rangle - \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq \langle\!\langle \boldsymbol{d}_8 \rangle\!\rangle$ simultaneously hold. Furthermore, let η denote a formula that encodes the equality $\langle\!\langle \boldsymbol{r0}' \rangle\!\rangle = \langle\!\langle \boldsymbol{d}'_1 \rangle\!\rangle$ where $\boldsymbol{d}'_1$ is a signed bit-vector representing d'_1. Presenting the compound formula $\kappa \wedge \varphi \wedge \eta$ to a SAT solver produces a model

$$\mathbf{m}_1 = \left\{ \; \langle\!\langle \boldsymbol{d}'_1 \rangle\!\rangle = 1, \quad \langle\!\langle \boldsymbol{d}_1 \rangle\!\rangle = 1, \quad \ldots, \langle\!\langle \boldsymbol{d}_8 \rangle\!\rangle = 1 \quad \right\}$$

that is fully detailed in Tab. 5.1. The assignment $\langle\!\langle \boldsymbol{d}'_1 \rangle\!\rangle = 1$ does not necessarily represent the maximum value of $\langle\!\langle \boldsymbol{d}'_1 \rangle\!\rangle$ for the partial assignment $\langle\!\langle \boldsymbol{d}_1 \rangle\!\rangle = 1, \ldots, \langle\!\langle \boldsymbol{d}_8 \rangle\!\rangle = 1$. Let ζ_1 thus denote a formula that holds iff $\langle\!\langle \boldsymbol{d}_1 \rangle\!\rangle = 1, \ldots, \langle\!\langle \boldsymbol{d}_8 \rangle\!\rangle = 1$ all hold. Then, dichotomic search can be applied to find the maximal value of $\langle\!\langle \boldsymbol{d}'_1 \rangle\!\rangle$ subject to $\kappa \wedge \varphi \wedge \eta \wedge \zeta$. This gives $\langle\!\langle \boldsymbol{d}'_1 \rangle\!\rangle = 2$ and a model:

$$\mathbf{m}'_1 \;=\; \{\langle\!\langle \boldsymbol{d}'_1 \rangle\!\rangle = 2, \langle\!\langle \boldsymbol{d}_1 \rangle\!\rangle = 1, \ldots, \langle\!\langle \boldsymbol{d}_8 \rangle\!\rangle = 1\}$$

An affine summary of all such maximal models can be found by interleaving range refinement with affine join $\sqcup_{\mathsf{aff}}$. Thus suppose the matrix $\mathbf{M}_1$ is constructed from $\mathbf{m}'_1$ by using the variable ordering $(d'_1, d_1, \ldots, d_8)$ on columns:

$$\mathbf{M}_1 = \left[\begin{array}{ccccccccc|c} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{array}\right]$$

The method proceeds in an analogous fashion to $\alpha^{\boldsymbol{V}}_{\mathsf{aff}}$ by constructing a formula μ that holds iff $\langle\!\langle \boldsymbol{d}_8 \rangle\!\rangle \neq 1$ holds, thereby violating the last row of $\mathbf{M}_1$. Solving the formula $\kappa \wedge \varphi \wedge \eta \wedge \mu$ gives the model $\mathbf{m}_2$ detailed in Tab. 5.1. The model $\mathbf{m}_2$ itself defines a formula ζ_2 that is equivalent to the conjunction of $\langle\!\langle \boldsymbol{d}_1 \rangle\!\rangle = 3, \ldots, \langle\!\langle \boldsymbol{d}_8 \rangle\!\rangle = 0$. Maximizing $\langle\!\langle \boldsymbol{d}'_1 \rangle\!\rangle$ subject to $\kappa \wedge \varphi \wedge \eta \wedge \zeta_2$ gives $\langle\!\langle \boldsymbol{d}'_1 \rangle\!\rangle = 10$, which defines the model

$$\mathbf{m}'_2 \;=\; \{\langle\!\langle \boldsymbol{d}'_1 \rangle\!\rangle = 10, \langle\!\langle \boldsymbol{d}_1 \rangle\!\rangle = 3, \ldots, \langle\!\langle \boldsymbol{d}_8 \rangle\!\rangle = 0\}$$

and $\mathbf{M}_2$, which in turn yields the join $\mathbf{M}_1 \sqcup_{\mathsf{aff}} \mathbf{M}_2$ as follows:

$$\mathbf{M}_1 \sqcup \mathbf{M}_2 = \left[\begin{array}{ccccccccc|c} 1 & 0 & 0 & 0 & 0 & -2 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{array}\right]$$

Repeating this process two more times then gives:

$$\begin{array}{rcl} \mathbf{m}'_3 & = & \{\langle\!\langle \boldsymbol{d}'_1 \rangle\!\rangle = 26, \langle\!\langle \boldsymbol{d}_1 \rangle\!\rangle = 8, \ldots, \langle\!\langle \boldsymbol{d}_8 \rangle\!\rangle = 0\} \\ \mathbf{m}'_4 & = & \{\langle\!\langle \boldsymbol{d}'_1 \rangle\!\rangle = 6, \langle\!\langle \boldsymbol{d}_1 \rangle\!\rangle = 0, \ldots, \langle\!\langle \boldsymbol{d}_8 \rangle\!\rangle = 3\} \end{array}$$

$$\mathbf{M}_1 \sqcup_{\mathsf{aff}} \mathbf{M}_2 \sqcup_{\mathsf{aff}} \mathbf{M}_3 = \left[\begin{array}{ccccccccc|c} 1 & 0 & 0 & 0 & 0 & -2 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \end{array}\right]$$

$$\mathbf{M}_1 \sqcup_{\mathsf{aff}} \mathbf{M}_2 \sqcup_{\mathsf{aff}} \mathbf{M}_3 \sqcup_{\mathsf{aff}} \mathbf{M}_4 = \left[\begin{array}{ccccccccc|c} 1 & 0 & 0 & 0 & 0 & -2 & 0 & 0 & 0 & 0 \end{array}\right]$$

Then, $\mathbf{M}_1 \sqcup \mathbf{M}_2 \sqcup \mathbf{M}_3 \sqcup \mathbf{M}_4$ expresses the relationship $\langle\!\langle \boldsymbol{d}'_1 \rangle\!\rangle = 2 \cdot \langle\!\langle \boldsymbol{d}_5 \rangle\!\rangle$. In summary, each iteration k of the algorithm involves the following steps:

1. Find a model of $\kappa \wedge \varphi \wedge \eta \wedge \mu$ where μ ensures that the model is not already summarized by $\sqcup_{i=1}^{k-1} \mathbf{M}_i$.
2. Maximize $\langle\!\langle \boldsymbol{d}'_1 \rangle\!\rangle$ whilst keeping $\langle\!\langle \boldsymbol{d}_1 \rangle\!\rangle, \ldots, \langle\!\langle \boldsymbol{d}_8 \rangle\!\rangle$ fixed.
3. Join the resulting model with $\sqcup_{i=1}^{k-1} \mathbf{M}_i$ to give $\sqcup_{i=1}^{k} \mathbf{M}_i$.

To verify that $\langle\!\langle \boldsymbol{d}'_1 \rangle\!\rangle = 2 \cdot \langle\!\langle \boldsymbol{d}_5 \rangle\!\rangle$ is a fixed point, unlike before, it is not sufficient to impose the disequality $\langle\!\langle \boldsymbol{d}'_1 \rangle\!\rangle \neq 2 \cdot \langle\!\langle \boldsymbol{d}_5 \rangle\!\rangle$ and check for unsatisfiability. This is because $\langle\!\langle \boldsymbol{d}'_1 \rangle\!\rangle$ is defined through maximization, rather than equality. Instead, the check amounts to testing whether $\kappa \wedge \varphi \wedge \eta$ is unsatisfiable when combined with a formula encoding $\langle\!\langle \boldsymbol{d}'_1 \rangle\!\rangle > 2 \cdot \langle\!\langle \boldsymbol{d}_5 \rangle\!\rangle$. Note that if $\langle\!\langle \boldsymbol{d}'_1 \rangle\!\rangle > 2 \cdot \langle\!\langle \boldsymbol{d}_5 \rangle\!\rangle$ holds, then it follows that $\langle\!\langle \boldsymbol{d}'_1 \rangle\!\rangle \neq 2 \cdot \langle\!\langle \boldsymbol{d}_5 \rangle\!\rangle$ holds. Since the combined system is unsatisfiable, we conclude that the update includes $d'_1 = 2 \cdot d_5$. The complete affine update consists of:

$$\begin{array}{lcllcl} d'_1 & = & 2 \cdot d_5 & \qquad d'_5 & = & 2 \cdot d_5 + d_2 \\ d'_2 & = & d_2 & d'_6 & = & 2 \cdot d_6 + d_4 \\ d'_3 & = & 2 \cdot d_6 & d'_7 & = & 2 \cdot d_6 + d_2 \\ d'_4 & = & d_4 & d'_8 & = & 2 \cdot d_5 + d_4 \end{array}$$

This result is superior to that computed in Ex. 5.3. To illustrate, consider again an input defined by $0 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq 4$, $0 \leq \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 1$ and $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 4$, hence:

$$d_1 = 4 \qquad d_2 = 1 \qquad d_3 = 0 \qquad d_4 = 0 \qquad d_5 = 4$$

Applying the computed transformer to derive d'_5 on output gives $d'_5 = 2 \cdot 4 + 1 = 9$. Hence, we have $\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1'} \rangle\!\rangle \leq 9$, whereas the previously discussed technique based on applying the β map from Def. 5.1 yields $\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1'} \rangle\!\rangle \leq 11$.

Correctness and Optimality

Indeed, these symbolic update operations are optimal in the sense that if a symbolic output constant d'_j is equal to a linear function of the symbolic input constants $d_1, \ldots, d_8$, then that function will be derived. In the discussion of Miné [166, Fig. 27], this technique might thus be classified as exact. The following theorem confirms this intuition. For ease of presentation, the result states the exactitude of the update on the constant d'_1. Analogous results hold for updates on $d'_2, \ldots, d'_8$.

Theorem 5.1. *Suppose an octagonal update of the form* $\mathbf{M} \in \mathbb{Q}^{10\times 1}$ *is derived such that* $\mathbf{M} \cdot (d_1', d_1, \ldots, d_8, -1)^T = 0$*. Moreover, suppose that:*

- *For all values of* $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle$ *and* $\langle\!\langle \boldsymbol{r1} \rangle\!\rangle$ *such that* $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq d_1$*,* $\ldots$*,* $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle - \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq d_8$ *and* φ *hold, it follows that* $\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle \leq c + c_1 \cdot d_1 + \ldots + c_8 \cdot d_8$ *holds.*
- *For all values of* $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle, \langle\!\langle \boldsymbol{r1} \rangle\!\rangle$*, there exists a value of* $\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle$ *such that* $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq d_1$*,* $\ldots$*,* $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle - \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq d_8$*,* φ *and* $\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = c + c_1 \cdot d_1 + \ldots + c_8 \cdot d_8$ *hold.*

Then, $\mathbf{M} \cdot (d_1', d_1, \ldots, d_8, -1)^T = 0$ *entails* $d_1' = c + c_1 \cdot d_1 + \ldots + c_8 \cdot d_8$

Proof. Suppose that $\mathbf{M}$ is derived by $\mathbf{M} = \mathbf{M}_1 \sqcup \mathbf{M}_2 \sqcup \ldots \sqcup \mathbf{M}_k$. Further, suppose $\mathbf{M}_1$ is constructed from the model $\mathbf{m}_1 = \{d_1' = v_1', d_1 = v_1, \ldots, d_8 = v_8\}$ where the value v_1' is maximal. Yet, $\mathbf{m}_1$ is derived from a formula that encodes the equality $\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = \langle\!\langle \boldsymbol{d_1'} \rangle\!\rangle$ where $\boldsymbol{d_1'}$ is a signed bit-vector representing d_1'. Since v_1' is maximal, it follows that the value of $\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle$ is maximal, too. Hence, it follows that $(d_1' = v_1') \wedge \bigwedge_{i=1}^{8} (d_i = v_i)$ implies $d_1' = c + c_1 \cdot d_1 + \ldots + c_8 \cdot d_8$ by the two assumptions. Therefore, $\mathbf{M}_1 \cdot (d_1', d_1, \ldots, d_8, -1)^T = 0$ entails $d_1' = c + c_1 \cdot d_1 + \ldots + c_8 \cdot d_8$ since $\mathbf{M}_1 = [I \mid (v_1', v_1, \ldots, v_8)^T]$, where $I \in \mathbb{Q}^{9\times 9}$ denotes the identity matrix. Likewise, $\mathbf{M}_i \cdot (d_1', d_1, \ldots, d_8, -1)^T = 0$ implies $d_1' = c + c_1 \cdot d_1 + \ldots + c_8 \cdot d_8$ for all $1 \leq i \leq k$. The result follows since $\mathbf{M}$ is the least upper bound of $\mathbf{M}_1, \mathbf{M}_2, \ldots, \mathbf{M}_k$, whereas $d_1' = c + c_1 \cdot d_1 + \ldots + c_8 \cdot d_8$ is an upper bound. □

Suppose $\mathbf{M}$ describes a symbolic bound d' of an output octagon as a linear combination of symbolic inputs $d_1, \ldots, d_k$. Since affine equalities over $d', d_1, \ldots, d_k$ constitute an ascending chain over $k+1$ variables, $\mathbf{M}$ is derived in at most $k+2$ iterations of affine abstraction (cp. Cor. 4.4). In each iteration, maximization is performed, which requires w' calls to a SAT solver, where $w' = w + 2$ (cp. Cor. 4.2). Hence, deriving an optimal affine description of d' using $d_1, \ldots, d_k$ requires $(k+2) \cdot w'$ calls to a SAT solver in the worst case.

5.3.2 Inferring Polynomial Equalities for Octagons

The minimization and maximization terms that arise during the evaluation of interval terms suggest a tactic for lifting polynomial updates to octagons. To illustrate with the update from Ex. 4.13, our construction proceeds by introducing a set $\mathcal{S} = \{d_1 \cdot d_3, d_1 \cdot d_6, d_3 \cdot d_5, d_5 \cdot d_6\}$ of monomials over the symbolic bounds of an octagon over $\boldsymbol{r0}$, $\boldsymbol{r1}$, and $\boldsymbol{r2}$. We then introduce two auxiliary variables p_1 and p_2 specified as:

$$p_1 \;=\; \max\{s \in \mathcal{S} \cup Z\} \qquad p_2 \;=\; \min\{s \in \mathcal{S} \cup Z\}$$

Here, Z is as defined in Def. 5.4. The goal of the analysis is then to infer an equality

$$d_i' \;=\; \textstyle\sum_{i=1}^{18} \lambda_i \cdot d_i + \rho_1 \cdot s_1 + \rho_2 \cdot s_2 + c$$

to specify each symbolic output d_i' as an affine combination of the inputs d_i and the non-linear terms s_1 and s_2. By interleaving the affine join over $\{d_i', d_1, \ldots, d_8, s_1, s_2\}$ with maximization as before, we obtain such an update. In this case, we compute the following transfer function:

$$\left\{\begin{array}{rcl}
\langle\!\langle \boldsymbol{r0} \rangle\!\rangle & \leq & d_1 \\
\langle\!\langle \boldsymbol{r1} \rangle\!\rangle & \leq & d_2 \\
\langle\!\langle \boldsymbol{r2} \rangle\!\rangle & \leq & d_3 \\
-\langle\!\langle \boldsymbol{r0} \rangle\!\rangle & \leq & d_4 \\
-\langle\!\langle \boldsymbol{r1} \rangle\!\rangle & \leq & d_5 \\
-\langle\!\langle \boldsymbol{r2} \rangle\!\rangle & \leq & d_6 \\
\hline
\langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle & \leq & d_7 \\
-\langle\!\langle \boldsymbol{r0} \rangle\!\rangle - \langle\!\langle \boldsymbol{r1} \rangle\!\rangle & \leq & d_8 \\
-\langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle & \leq & d_9 \\
\langle\!\langle \boldsymbol{r0} \rangle\!\rangle - \langle\!\langle \boldsymbol{r1} \rangle\!\rangle & \leq & d_{10} \\
\hline
\langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r2} \rangle\!\rangle & \leq & d_{11} \\
-\langle\!\langle \boldsymbol{r0} \rangle\!\rangle - \langle\!\langle \boldsymbol{r2} \rangle\!\rangle & \leq & d_{12} \\
-\langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r2} \rangle\!\rangle & \leq & d_{13} \\
\langle\!\langle \boldsymbol{r0} \rangle\!\rangle - \langle\!\langle \boldsymbol{r2} \rangle\!\rangle & \leq & d_{14} \\
\hline
\langle\!\langle \boldsymbol{r1} \rangle\!\rangle + \langle\!\langle \boldsymbol{r2} \rangle\!\rangle & \leq & d_{15} \\
-\langle\!\langle \boldsymbol{r1} \rangle\!\rangle - \langle\!\langle \boldsymbol{r2} \rangle\!\rangle & \leq & d_{16} \\
-\langle\!\langle \boldsymbol{r1} \rangle\!\rangle + \langle\!\langle \boldsymbol{r2} \rangle\!\rangle & \leq & d_{17} \\
\langle\!\langle \boldsymbol{r1} \rangle\!\rangle - \langle\!\langle \boldsymbol{r2} \rangle\!\rangle & \leq & d_{18}
\end{array}\right\}
\rightsquigarrow
\left\{\begin{array}{rcl}
\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle & \leq & d_2 + s_1 \\
\langle\!\langle \boldsymbol{r1'} \rangle\!\rangle & \leq & d_2 \\
\langle\!\langle \boldsymbol{r2'} \rangle\!\rangle & \leq & d_3 \\
-\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle & \leq & d_5 + s_2 \\
-\langle\!\langle \boldsymbol{r1'} \rangle\!\rangle & \leq & d_5 \\
-\langle\!\langle \boldsymbol{r2'} \rangle\!\rangle & \leq & d_6 \\
\hline
\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1'} \rangle\!\rangle & \leq & d_2 + s_1 + d_2 \\
-\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle - \langle\!\langle \boldsymbol{r1'} \rangle\!\rangle & \leq & d_5 + s_2 + d_5 \\
-\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1'} \rangle\!\rangle & \leq & d_5 + s_2 + d_2 \\
\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle - \langle\!\langle \boldsymbol{r1'} \rangle\!\rangle & \leq & d_2 + s_1 + d_5 \\
\hline
\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle + \langle\!\langle \boldsymbol{r2'} \rangle\!\rangle & \leq & d_2 + s_1 + d_3 \\
-\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle - \langle\!\langle \boldsymbol{r2'} \rangle\!\rangle & \leq & d_5 + s_2 + d_6 \\
-\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle + \langle\!\langle \boldsymbol{r2'} \rangle\!\rangle & \leq & d_5 + s_2 + d_3 \\
\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle - \langle\!\langle \boldsymbol{r2'} \rangle\!\rangle & \leq & d_2 + s_1 + d_6 \\
\hline
\langle\!\langle \boldsymbol{r1'} \rangle\!\rangle + \langle\!\langle \boldsymbol{r2'} \rangle\!\rangle & \leq & d_{15} \\
-\langle\!\langle \boldsymbol{r1'} \rangle\!\rangle - \langle\!\langle \boldsymbol{r2'} \rangle\!\rangle & \leq & d_{16} \\
-\langle\!\langle \boldsymbol{r1'} \rangle\!\rangle + \langle\!\langle \boldsymbol{r2'} \rangle\!\rangle & \leq & d_{17} \\
\langle\!\langle \boldsymbol{r1'} \rangle\!\rangle - \langle\!\langle \boldsymbol{r2'} \rangle\!\rangle & \leq & d_{18}
\end{array}\right\}$$

Thus, for example, if the octagon on input describes a cube that is offset from the origin, namely, $d_1 = d_2 = d_3 = 3$ and $d_4 = d_5 = d_6 = -2$, then the bound on $\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle$, denoted d_1', is calculated by:

$$d_1' \quad = \quad d_2 + s_1 \quad = \quad 3 + \max\{3 \cdot 3, 3 \cdot (-2), (-2) \cdot 3, (-2) \cdot -2\} \quad = \quad 12$$

Compared to the lifting technique in Chap. 5.1.4, the output is in general superior. This is because the derived transformer uses monomials over arbitrary symbolic constants d_i on input, rather than merely the range constraints, though this is not the case in the above example.

5.3.3 Optimal Affine Updates on Arithmetical Congruences

Thus far, the domain of arithmetical congruences was only touched in Chap. 4.2.4. There, we have presented a procedure $\alpha^{\boldsymbol{V}}_{\mathsf{a\text{-}cong}}(\varphi)$ that computes characterizations of the form $\langle \boldsymbol{v} \rangle \equiv_m c$ for all bit-vectors $\boldsymbol{v} \in \boldsymbol{V}$. A related question in the context of transfer function synthesis is thus how affine or polynomial updates can be

inferred for symbolic representations of arithmetical congruences. Specifically, we face the following question: Given symbolic representations $\langle \boldsymbol{v}_i \rangle \equiv_{m_i} c_i$ for all inputs $\boldsymbol{v}_i \in \boldsymbol{V}_{\mathsf{in}}$, how can the parameters m' and c' of an arithmetical congruence $\langle \boldsymbol{v}' \rangle \equiv_{m'} c'$ of an output bit-vector $\boldsymbol{v}' \in \boldsymbol{V}_{\mathsf{out}}$ be characterized as an affine (or polynomial) formula over $m_1, c_1, \ldots, m_n, c_n$?

Affine Relations over Arithmetical Congruences

To illustrate the algorithm, assume an 8-bit instruction `NOT R0` in regular mode, which computes the bit-wise negation of `R0` and stores the result in `R0`. The semantics of `NOT R0` is represented by a formula $\varphi \in \wp(\wp(\boldsymbol{V}_{\mathsf{in}} \cup \boldsymbol{V}_{\mathsf{out}}))$ with $\boldsymbol{V}_{\mathsf{in}} = \{\boldsymbol{r0}\}$ and $\boldsymbol{V}_{\mathsf{out}} = \{\boldsymbol{r0'}\}$. We introduce bit-vectors $\boldsymbol{m_{r0}}$, $\boldsymbol{c_{r0}}$, $\boldsymbol{m_{r0'}}$, and $\boldsymbol{c_{r0'}}$. Let κ encode the requirement that the arithmetical congruences $\langle \boldsymbol{r0} \rangle \equiv_{\boldsymbol{m_{r0}}} \langle \boldsymbol{c_{r0}} \rangle$ and $\langle \boldsymbol{r0'} \rangle \equiv_{\boldsymbol{m_{r0'}}} \langle \boldsymbol{c_{r0'}} \rangle$ simultaneously hold. Computing an affine equality that describes $\mathbf{m}_{\boldsymbol{r0'}}$ is then straightforward, as this iteration amounts to representing the values of $\boldsymbol{m_{r0'}}$, $\boldsymbol{m_{r0}}$, and $\boldsymbol{c_{r0}}$ as a matrix and then iteratively computing the affine hull, as before. The iteration eventually stabilizes with $\langle \boldsymbol{m_{r0'}} \rangle = 255 - \langle \boldsymbol{m_{r0}} \rangle$. Applying the same strategy to $\langle \boldsymbol{c_{r0'}} \rangle$ yields $\langle \boldsymbol{c_{r0'}} \rangle = \langle \boldsymbol{c_{r0}} \rangle$. We thus obtain the overall transformer $\langle \boldsymbol{r0'} \rangle \equiv_{255-\langle \boldsymbol{m_{r0}} \rangle} \langle \boldsymbol{c_{r0}} \rangle$, which is the optimal result.

Probabilistic Affine Equalities for Arithmetical Congruences

Unfortunately, computing descriptive results for the displacement $\langle \boldsymbol{c_{r0}} \rangle$ is not always as easy. For example, a transformer for `ADD R0 R1` in regular mode involves the computation of the greatest common divisor of the displacements of the congruences that describe `R0` and `R1`. If $\langle \boldsymbol{r0} \rangle \equiv_{\langle \boldsymbol{m_{r0}} \rangle} \langle \boldsymbol{c_{r0}} \rangle$ and $\langle \boldsymbol{r1} \rangle \equiv_{\langle \boldsymbol{m_{r1}} \rangle} \langle \boldsymbol{c_{r1}} \rangle$ on input, then:

$$\langle \boldsymbol{r0'} \rangle \quad \equiv_{\langle \boldsymbol{m_{r0}} \rangle + \langle \boldsymbol{m_{r1}} \rangle} \quad \mathtt{gcd}(\langle \boldsymbol{c_{r0}} \rangle, \langle \boldsymbol{c_{r1}} \rangle)$$

The difficult part in deriving such a relation is to formalize $\mathtt{gcd}(\langle \boldsymbol{c_{r0}} \rangle, \langle \boldsymbol{c_{r1}} \rangle)$ within the SAT instance. Since $\mathtt{gcd}(a, b)$ can be computed in $\mathcal{O}(\mathsf{ln}(b))$, the term $\mathtt{gcd}(a, b)$ can be encoded in SAT by unrolling the extended Euclidean algorithm. However, it is also possible to sidestep this problem using a simple trick, although at the cost of soundness. Since a generic transformer is parametric in the modulus, we can define certain, concrete values of $\langle \boldsymbol{m_{r0}} \rangle$ and $\langle \boldsymbol{m_{r1}} \rangle$, e.g., $\langle \boldsymbol{m_{r0}} \rangle = 8$ and $\langle \boldsymbol{m_{r1}} \rangle = 11$, which entails $0 \leq \langle \boldsymbol{c_{r0}} \rangle < 8$ and $0 \leq \langle \boldsymbol{c_{r1}} \rangle < 11$. To do so, we define:

$$\mu \quad = \quad \begin{cases} (\langle \boldsymbol{m_{r0}} \rangle = 8) & \wedge & (\langle \boldsymbol{m_{r1}} \rangle = 11) & \wedge \\ (\langle \boldsymbol{m_{r0}} \rangle = 8) & \wedge & (\langle \boldsymbol{m_{r1}} \rangle = 11) & \wedge \\ (\langle \boldsymbol{c_{r0}} \rangle < 8) & \wedge & (\langle \boldsymbol{c_{r1}} \rangle < 11) & \end{cases}$$

Based on these values encoded in μ, we introduce symbolic variables that represent commonly occurring expressions such as $\mathtt{gcd}(\langle \boldsymbol{c_{r0}} \rangle, \langle \boldsymbol{c_{r1}} \rangle)$ and $\mathtt{lcm}(\langle \boldsymbol{c_{r0}} \rangle, \langle \boldsymbol{c_{r1}} \rangle)$

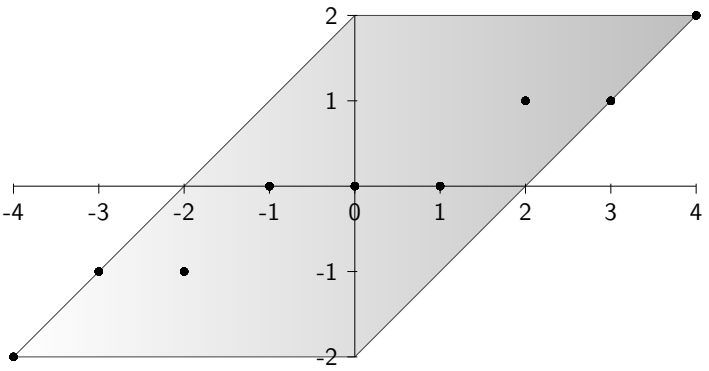


Figure 5.2: Octagon describing an assignment `y = x div 2` subject to $-4 \leq \mathtt{x} \leq 4$

drawn from a set of templates. In this example, we stick to $\mathtt{gcd}(\langle \boldsymbol{c_{r0}} \rangle, \langle \boldsymbol{c_{r1}} \rangle)$. The solutions of $\mathtt{gcd}(\langle \boldsymbol{c_{r0}} \rangle, \langle \boldsymbol{c_{r1}} \rangle)$ are then simply enumerated as part of the propositional formula. Let ν denote the valuations of the symbolic expression $p = \mathtt{gcd}(\langle \boldsymbol{c_{r0}} \rangle, \langle \boldsymbol{c_{r1}} \rangle)$ for $(\langle \boldsymbol{c_{r0}} \rangle < 8) \wedge (\langle \boldsymbol{c_{r1}} \rangle < 11)$. Then, computing the affine hull of

$$\{\boldsymbol{m_{r0}}, \boldsymbol{c_{r0}}, \boldsymbol{m_{r1}}, \boldsymbol{c_{r1}}, \boldsymbol{m_{r0'}}, \boldsymbol{c_{r0'}}, p\}$$

subject to $\varphi \wedge \mu \wedge \nu$ gives $\langle \boldsymbol{r0'} \rangle \equiv_{\langle \boldsymbol{m_{r0}} \rangle + \langle \boldsymbol{m_{r1}} \rangle} \mathtt{gcd}(\langle \boldsymbol{c_{r0}} \rangle, \langle \boldsymbol{c_{r1}} \rangle)$ as desired. Of course, this transformer may be valid only for the case that $\langle \boldsymbol{m_{r0}} \rangle = 8$ and $\langle \boldsymbol{m_{r1}} \rangle = 11$. However, we can increase the probability of (and confidence in) correctness of the abstraction by repeatedly applying the method for different coprime values of $\langle \boldsymbol{m_{r0}} \rangle$ and $\langle \boldsymbol{m_{r1}} \rangle$. In our implementation, the analysis is performed with 16 different valuations of the input moduli. We have never observed a faulty abstraction using this strategy. The method thus dovetails with the key idea behind *random interpretation* [123, 124], which is based on iterative simulation of programs to guarantee affine invariants that hold with high probability.

5.4 Affine Transformers for Non-Affine Relations

Octagonal constraints exhibit restricted capabilities for extracting affine transformers for range analysis of non-linear programs. Indeed, inequalities of the form $\pm\langle\!\langle \boldsymbol{v}_i \rangle\!\rangle \pm \langle\!\langle \boldsymbol{v}_j \rangle\!\rangle \leq d$ can be turned into a transformer that over-approximates the ranges of either $\langle\!\langle \boldsymbol{v}_i \rangle\!\rangle$ or $\langle\!\langle \boldsymbol{v}_j \rangle\!\rangle$. As an example, consider `y = x div 2` subject to the additional constraints $-4 \leq \mathtt{x} \leq 4$. The optimal abstraction using octagons, which defines

$$(-4 \leq \mathtt{x} \leq 4) \quad (-2 \leq \mathtt{y} \leq 2) \quad (-2 \leq \mathtt{x} - \mathtt{y} \leq 2)$$

is given in Fig. 5.2 (the redundant constraints for $\mathtt{x}+\mathtt{y}$ and $-\mathtt{x}-\mathtt{y}$ have been omitted). From the third constraint, we obtain inequalities $\mathtt{y} \leq \mathtt{x}+2$ and $\mathtt{y} \geq \mathtt{x}-2$, which can be used to express ranges on $\mathtt{y}$. If $\mathtt{x}_\ell$, $\mathtt{x}_u$, $\mathtt{y}_\ell$, and $\mathtt{y}_u$ denote the extremal values of $\mathtt{x}$ and $\mathtt{y}$, respectively, the octagon defines an over-approximate update for interval analysis:

$$\mathtt{y}_\ell \;=\; \mathtt{x}_\ell - 2 \qquad \mathtt{y}_u \;=\; \mathtt{x}_u + 2$$

Observe that a conjunction of two-variables-per-inequality (TVPI) constraints [224, 225], which again forms a convex shape, would compute a better description of the same set of solutions. The TVPI abstraction consists of the following six inequalities:

$$\begin{pmatrix} \mathtt{y} & \leq & \mathtt{x}+2 \\ \mathtt{y} & \leq & \frac{1}{2}\cdot\mathtt{x}+\frac{1}{2} \\ \mathtt{y} & \leq & \frac{4}{3}\cdot\mathtt{x}+\frac{4}{3} \end{pmatrix} \;\bigwedge\; \begin{pmatrix} \mathtt{y} & \geq & \frac{2}{5}\cdot\mathtt{x}-\frac{2}{5} \\ \mathtt{y} & \geq & \frac{1}{2}\cdot\mathtt{x}-\frac{1}{2} \\ \mathtt{y} & \geq & \mathtt{x}-2 \end{pmatrix}$$

Indeed, the system of TVPI constraints does not contain any spurious integral solutions to the assignment `y = x div 2`, which compares favorably against the octagonal system. We generate the following transformers for interval analysis from the TVPI abstraction, applying the lifting techniques introduced before:

$$\begin{array}{rcl} \mathtt{y}_\ell & = & \lceil \max\left\{\frac{2}{5}\cdot\mathtt{x}_\ell - \frac{2}{5}, \frac{1}{2}\cdot\mathtt{x}_\ell - \frac{1}{2}, \mathtt{x}_\ell - 2\right\}\rceil \\ \mathtt{y}_u & = & \lfloor \min\left\{\mathtt{x}_u + 2, \frac{1}{2}\cdot\mathtt{x}_u + \frac{1}{2}, \frac{2}{5}\cdot\mathtt{x}_u + \frac{2}{5}\right\}\rfloor \end{array}$$

For inputs $\mathtt{x}_\ell = 0$ and $\mathtt{x}_u = 1$, the TVPI abstraction thus induces the output:

$$\begin{array}{rcl} \mathtt{x}_\ell & = & \lceil \max\left\{\frac{2}{5}\cdot 0 - \frac{2}{5}, \frac{1}{2}\cdot 0 - \frac{1}{2}, 0 - 2\right\}\rceil \;=\; 0 \\ \mathtt{y}_u & = & \lfloor \min\left\{1 + 2, \frac{1}{2}\cdot 1 + \frac{1}{2}, \frac{2}{5}\cdot 1 + \frac{2}{5}\right\}\rfloor \;=\; 0 \end{array}$$

By way of comparison, the octagon defines $\mathtt{y}_\ell = -2$ and $\mathtt{y}_u = 2$, which is, of course, non-optimal. The remainder of this section studies how to transform abstractions in terms of convex polyhedra into updates on intervals and octagons. Of course, the domain of convex polyhedra subsumes the TVPI domain employed in the example.

5.4.1 From Convex Polyhedra to Intervals

We assume that $\varphi \in \wp(\wp(\boldsymbol{V}))$ encodes the semantics of a block. Further, let $\boldsymbol{V}_{\mathsf{in}} \subseteq \boldsymbol{V}$ and $\boldsymbol{V}_{\mathsf{out}} \subseteq \boldsymbol{V}$ such that $\boldsymbol{V}_{\mathsf{in}} \cap \boldsymbol{V}_{\mathsf{out}} = \emptyset$ denote the bit-vectors on entry and exit of the respective block. Then, we compute $\alpha_{\mathsf{conv}}^{\boldsymbol{V}_{\mathsf{in}} \cup \{\boldsymbol{v}'\}}(\varphi)$ for each $\boldsymbol{v}' \in \boldsymbol{V}_{\mathsf{out}}$, which yields a convex polyhedron $c_{\boldsymbol{v}'}$ over $\boldsymbol{V}_{\mathsf{in}} \cup \{\boldsymbol{v}'\}$. We can straightforwardly transform each inequality so that $c_{\boldsymbol{v}'}$ takes the following shape:

$$\begin{array}{ll} \bigwedge_{j=1}^{m_1} & \left(\langle\!\langle \boldsymbol{v}' \rangle\!\rangle \geq \sum_{\boldsymbol{v} \in \boldsymbol{V}_{\mathsf{in}}} \lambda_{\boldsymbol{v},i} \cdot \langle\!\langle \boldsymbol{v} \rangle\!\rangle + c_i\right) \;\wedge \\ \bigwedge_{j=m_1+1}^{m_2} & \left(\langle\!\langle \boldsymbol{v}' \rangle\!\rangle \leq \sum_{\boldsymbol{v} \in \boldsymbol{V}_{\mathsf{in}}} \lambda_{\boldsymbol{v},j} \cdot \langle\!\langle \boldsymbol{v} \rangle\!\rangle + c_j\right) \end{array}$$

Then, the first m_1 inequalities characterize lower upper bounds on $\langle\!\langle \boldsymbol{v}' \rangle\!\rangle$, followed by m_2 inequalities that describe upper bounds. It is thus safe to deduce

$$\langle\!\langle \boldsymbol{v}'_\ell \rangle\!\rangle \;\leq\; \textstyle\sum_{\boldsymbol{v}\in\boldsymbol{V}_{\mathsf{in}}} \lambda_{\boldsymbol{v},i} \cdot \langle\!\langle \boldsymbol{v} \rangle\!\rangle + c_i \qquad \langle\!\langle \boldsymbol{v}'_u \rangle\!\rangle \;\leq\; \textstyle\sum_{\boldsymbol{v}\in\boldsymbol{V}_{\mathsf{in}}} \lambda_{\boldsymbol{v},j} \cdot \langle\!\langle \boldsymbol{v} \rangle\!\rangle + c_j$$

for all $1 \leq i \leq m_1$ and $m_1 + 1 \leq j \leq m_1 + m_2$. From monotonicity of $\sum_{\boldsymbol{v}\in\boldsymbol{V}_{\mathsf{in}}} \lambda_{\boldsymbol{v},i} \cdot \langle\!\langle \boldsymbol{v} \rangle\!\rangle + c_i$ and $\sum_{\boldsymbol{v}\in\boldsymbol{V}_{\mathsf{in}}} \lambda_{\boldsymbol{v},j} \cdot \langle\!\langle \boldsymbol{v} \rangle\!\rangle + c_j$, respectively, we deduce that lifting affine characterizations in $c_{\boldsymbol{v}'}$ to intervals becomes applicable (cp. Chap. 5.1.1). By applying Prop. 5.1, which we combine with floor and ceil operators to give integral results, we obtain:

$$\begin{array}{rcl} \langle\!\langle \boldsymbol{v}'_\ell \rangle\!\rangle & = & \lceil \max\{\sum_{\boldsymbol{v}\in\boldsymbol{V}_{\mathsf{in}}} \lambda_{\boldsymbol{v},i} \cdot \beta(-\lambda_{\boldsymbol{v},i}, \langle\!\langle \boldsymbol{v} \rangle\!\rangle) + c_i \mid 1 \leq i \leq m_1\} \rceil \\ \langle\!\langle \boldsymbol{v}'_u \rangle\!\rangle & = & \lfloor \min\{\sum_{\boldsymbol{v}\in\boldsymbol{V}_{\mathsf{in}}} \lambda_{\boldsymbol{v},j} \cdot \beta(\lambda_{\boldsymbol{v},j}, \langle\!\langle \boldsymbol{v} \rangle\!\rangle) + c_j \mid m_1 + 1 \leq j \leq m_1 + m_2\} \rfloor \end{array}$$

It is thus possible to derive an almost affine update — one that involves minimization and maximization during evaluation — for intervals directly from polyhedral abstractions of φ, without loss in precision.

5.4.2 From Convex Polyhedra to Octagons

When we discussed lifting of affine equalities to octagons in Chap. 5.1.2, we have transformed the unary constraints of an input octagon to characterize the output, thereby inducing a loss in precision. It is possible to derive octagonal transformers from polyhedral abstractions using this strategy, too. Suppose that $\alpha_{\mathsf{conv}}^{\boldsymbol{V}_{\mathsf{in}}\cup\{\boldsymbol{v}'_1\}}(\varphi)$ and $\alpha_{\mathsf{conv}}^{\boldsymbol{V}_{\mathsf{in}}\cup\{\boldsymbol{v}'_2\}}(\varphi)$ for $\boldsymbol{v}'_1, \boldsymbol{v}'_2 \in \boldsymbol{V}_{\mathsf{out}}$ yield polyhedral descriptions of $\langle\!\langle \boldsymbol{v}'_1 \rangle\!\rangle$ and $\langle\!\langle \boldsymbol{v}'_2 \rangle\!\rangle$, respectively. For unary constraints (such as $\langle\!\langle \boldsymbol{v}'_1 \rangle\!\rangle \leq d'_1$), we apply the strategy for intervals, which gives an update on d'_1 that involves minimization and maximization. For the more involved case of binary constraints of the form $\lambda_1 \cdot \langle\!\langle \boldsymbol{v}'_1 \rangle\!\rangle + \lambda_2 \cdot \langle\!\langle \boldsymbol{v}'_2 \rangle\!\rangle \leq d'$, we proceed as follows: put $d' = \lambda_1 \cdot \tau_1 + \lambda_2 \cdot \tau_2$ where τ_i is defined as:

$$\tau_i \;=\; \begin{cases} \text{lower bound of } \langle\!\langle \boldsymbol{v}'_i \rangle\!\rangle & : \; \lambda_i = -1 \\ \text{upper bound of } \langle\!\langle \boldsymbol{v}'_i \rangle\!\rangle & : \; \lambda_i = 1 \end{cases}$$

This transformation solely uses interval updates. We thus obtain binary output constraints using combinations of the inputs, though at the cost of precision.

5.4.3 Interleaving Polyhedral Abstraction and Maximization

Yet, it is also possible to interleave octagonal abstraction with polyhedral abstraction, thereby extending the technique in Chap. 5.3. The express aim of this technique is to derive a convex polyhedron that describes the constants d'_i of an octagon by a conjunction of linear inequalities that involve the constants d_i on input. As in Chap. 5.3, we proceed by introducing sign-extended bit vectors $\boldsymbol{d}_1, \ldots, \boldsymbol{d}_8$ to represent the

symbolic constants $d_1, \ldots, d_8$ of the input octagon. Further, let κ denote a Boolean formula that holds iff the eight inequalities $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq \langle\!\langle \boldsymbol{d}_1 \rangle\!\rangle, \ldots, \langle\!\langle \boldsymbol{r0} \rangle\!\rangle - \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq \langle\!\langle \boldsymbol{d}_8 \rangle\!\rangle$ simultaneously hold; likewise, let η encode the equality $\langle\!\langle \boldsymbol{r0}' \rangle\!\rangle = \langle\!\langle \boldsymbol{d}'_1 \rangle\!\rangle$ where $\boldsymbol{d}'_1$ is a signed bit-vector representing d'_1. In essence, the algorithm performs each of the following steps in each iteration k:

1. Find a model $\mathbf{m}_k$ of $\kappa \wedge \varphi \wedge \eta \wedge \mu$ where μ ensures that $\mathbf{m}$ is not already summarized by $\alpha^{\mathbb{Z}^9}_{\mathsf{conv}}(\mathbf{c}_1) \sqcup_{\mathsf{conv}} \ldots \sqcup_{\mathsf{conv}} \alpha^{\mathbb{Z}^9}_{\mathsf{conv}}(\mathbf{c}_{k-1})$. As before, $\alpha^{\mathbb{Z}^n}_{\mathsf{conv}}(c)$ denotes an operation that converts a point $c \in \mathbb{Z}^9$ into a convex polyhedron.

2. Maximize $\langle\!\langle \boldsymbol{d}'_1 \rangle\!\rangle$ whilst keeping $\langle\!\langle \boldsymbol{d}_1 \rangle\!\rangle = \mathbf{m}(\boldsymbol{d}_1), \ldots, \langle\!\langle \boldsymbol{d}_8 \rangle\!\rangle = \mathbf{m}(\boldsymbol{d}_8)$ fixed, which gives $d'_{1,k} \in \mathbb{Z}$.

3. Join $(d'_{1,k}, \mathbf{m}(\boldsymbol{d}_1), \mathbf{m}(\boldsymbol{d}_8))$ with $\alpha^{\mathbb{Z}^9}_{\mathsf{conv}}(\mathbf{c}_1) \sqcup_{\mathsf{conv}} \ldots \sqcup_{\mathsf{conv}} \alpha^{\mathbb{Z}^9}_{\mathsf{conv}}(\mathbf{c}_{k-1})$.

Upon termination, the convex polyhedron $\alpha^{\mathbb{Z}^9}_{\mathsf{conv}}(\mathbf{c}_1) \sqcup_{\mathsf{conv}} \ldots \sqcup_{\mathsf{conv}} \alpha^{\mathbb{Z}^9}_{\mathsf{conv}}(\mathbf{c}_k)$ over-approximates d'_1 in the sense that d'_1 is characterized by a conjunction of linear inequalities over $d_1, \ldots, d_8$. As in Chap. 5.3, the technique can repeatedly be applied to characterize $d'_2, \ldots, d'_8$, too.

5.5 Experiments

As in Chap. 4.4, we report on experimental results that we have obtained using a prototype implementation written in C++ on top of Z3.

5.5.1 Lifting and Transformation

The computational cost of lifting affine or polynomial constraints to intervals and octagon, i.e., rewriting equalities $\boldsymbol{V}_{\mathsf{in}} \xrightarrow{\mathsf{Aff}} \boldsymbol{V}_{\mathsf{out}}$ and $\boldsymbol{V}_{\mathsf{in}} \xrightarrow{\mathsf{Poly}} \boldsymbol{V}_{\mathsf{out}}$ as

$$\boldsymbol{V}_{\mathsf{in}} \xrightarrow{\mathsf{Aff}} \boldsymbol{V}_{\mathsf{out}} \quad \rightsquigarrow \quad \begin{cases} \mathsf{Int} \xrightarrow{\mathsf{Aff}} \boldsymbol{V}_{\mathsf{out}} \\ \mathsf{Oct} \xrightarrow{\mathsf{Aff}} \mathsf{Oct} \end{cases} \qquad \boldsymbol{V}_{\mathsf{in}} \xrightarrow{\mathsf{Poly}} \boldsymbol{V}_{\mathsf{out}} \quad \rightsquigarrow \quad \begin{cases} \mathsf{Int} \xrightarrow{\mathsf{Poly}} \mathsf{Int} \\ \mathsf{Oct} \xrightarrow{\mathsf{Poly}} \mathsf{Oct} \end{cases}$$

is negligible. The runtimes for each block are identical to those required for computing $\alpha^{\boldsymbol{V}}_{\mathsf{aff}}(\varphi)$ and $\alpha^{\boldsymbol{V}}_{\mathsf{poly}}(\varphi)$, which we have presented in Chap. 4.4.6 and Chap. 4.4.7. Likewise, the overhead of transforming polyhedra into updates on intervals (cp. Chap. 5.4.1) and octagons (cp. Chap. 5.4.2) is insignificant: The overall runtime is essentially that required for computing $\alpha^{\boldsymbol{V}}_{\mathsf{conv}}(\varphi)$. Therefore, we omit these numbers and refer the reader to Chap. 4.4.4.

Table 5.2: Transfer function synthesis using quantification

block	#instr.	$\mathbf{D_1}$	$\mathbf{D_2}$	#bits	time
ABS	5	Int	Int	8	0.6
				32	∞
INC	1	Int	Int	8	0.2
				32	15.8
INC&ASR	2	Int	Int	8	0.3
				32	18.3
SWAP	3	Int	Int	8	0.1
				32	0.2

5.5.2 Quantification

Table 5.2 presents experimental results for the application of the quantifier-based approach discussed in Chap. 5.2 to several blocks. In the table, the symbol ∞ indicates a timeout, which is set to 30 seconds. The result that was computed is a transformer of type $\mathsf{Int} \xrightarrow{\mathsf{Bool}} \mathsf{Int}$ for registers of width 8 and 32, respectively. Our experiences indicate that applying the quantifier-based approach to 8-bit registers is tractable. Yet, a slight increase in the bit-width can already entail intractability. We have not been able to obtain results for any except the simplest arithmetic operations (e.g., INC) for the 32-bit case in a reasonable amount of time. The block SWAP is interesting since it consists of three consecutive exclusive-or operations, for which there is no coupling between the different bits of the same register. This property makes it ideal for the quantifier-based approach, as quantifier elimination yields a very simple Boolean output formula. On average, however, approaches based on dichotomic search clearly outperform the quantifier-based approach.

5.5.3 Interleaved Abstraction

Comparing interleaved abstraction for octagons to the quantifier-based method, the former has predictable cost as the number of iterations required to converge onto an abstraction is linear in the number of octagonal constraints. To characterize a single output constraint $\pm\langle\!\langle \boldsymbol{v}'_1 \rangle\!\rangle \pm \langle\!\langle \boldsymbol{v}' \rangle\!\rangle \leq d'$, it is necessary to compute an ascending affine chain over d' and the symbolic bounds on input. Each iteration of the abstraction then involves maximization, which has predictable cost, too. Table 5.3 presents figures for several blocks (cp. Tab. 4.2) for all feasible mode combinations.

For the operation ADD R0 R1, e.g., an affine abstraction is computed that prescribes how an octagon over R0 and R1 on input is transformed into an octagon over the same variables on output. Thus, for 8 symbolic outputs, an affine equality is computed, which is interleaved with dichotomic search. Clearly, dichotomic search

Table 5.3: Interleaved abstraction using octagons and affine equalities

block	#instr.	#bits	time	block	#instr.	#bits	time
ABS	5	8	2.10	ADD&ASR	2	8	1.48
		16	5.35			16	3.80
		32	11.39			32	7.97
ADD	1	8	1.42	SWAP	3	8	1.20
		16	3.71			16	2.94
		32	8.03			32	5.88

has a strong impact on the overall runtime. Doubling the bit-width roughly doubles the runtime, which can be explained by twice the number of iterations required for dichotomic search. This effect suggests to apply extrapolation to this problem, too. Then, a candidate $d_i' = \sum_{i=1}^{8} \lambda_i \cdot d_i + c$ for an affine transformer for octagons over, say, 32-bit registers is derived from abstractions over short bit-vectors. Analyzing short bit-vectors entails a small number of iterations for dichotomic search, and thus, shorter runtimes. Checking soundness then amounts to testing $d_i' > \sum_{i=1}^{8} \lambda_i \cdot d_i + c$ for unsatisfiability. Overall, we have derived octagonal transformers for registers of width 32 and 64 using this approach within less than 2 seconds for each basic block from Tab. 4.2.

5.6 Related Work

The problem of designing transfer functions for numeric or symbolic domains is as old as the field of abstract interpretation itself [77]. Even the technique of using primed and unprimed variables to capture and abstract the semantics of program statements and functions dates back to the thesis work of Halbwachs [127]. Ideally, the abstract operations should compute abstractions that are as descriptive as possible, although there is usually interplay with accuracy and complexity (cp. [186]). However, even for a fixed abstract domain and a concrete program statement, there are typically many ways of designing and implementing transfer functions. Cousot and Halbwachs [82, Sect. 4.2.1], for example, discussed several ways to realize a transfer function for assignments such as $x = y \cdot z$ in the domain of convex polyhedra. Likewise, abstracting integer division $x = y/z$ (cp. Fig. 5.2) is an interesting study within itself [219]. The complexity of designing transfer functions of course depends not only on the concrete program statements, but also on the structure and complexity of the respective abstract domain. Examples of domains that necessitate sophisticated (and complex) abstract transformers include linear congruences, the difficulty of which was lamented by Granger [115], but also symbolic cache abstractions [117].

5.6.1 Generation of Symbolic Best Transformers

From the lamentation of Granger [115], however, it took more than a decade until it was observed that symbolic best transformers can always be found for abstract domains of finite height [197], provided one is prepared to pay the cost of repeatedly calling a decision procedure to evaluate a transformer at runtime. This strategy differs from our work. By way of contrast, our work aspires to evaluate transfer functions without a complicated decision procedure by computing it offline so as to both simplify and speedup their evaluation. Contemporaneously to Reps et al. [197], it was observed that best transformers for intervals can be computed using BDDs [186]. The authors observed that if $g : [0, 2^8 - 1] \to [0, 2^8 - 1]$ is a unary operation on an unsigned byte, then its best transformer $f : D \to D$ on $D = \bot_{\mathsf{int}} \cup \{[\ell, u] \mid 0 \leq l \leq u \leq 2^8 - 1\}$ can be defined through interval subdivision. If $\ell = u$, then $f([\ell, u]) = g(l)$; otherwise, if $\ell < u$, then $f([\ell, u]) = f([\ell, m - 1] \sqcup_{\mathsf{int}} f([m, u])$ where $m = \lfloor u/2^n \rfloor \cdot 2^n$ and $n = \lfloor \log_2(u - l + 1) \rfloor$. Binary operations can likewise be decomposed. The 8-bit inputs ℓ and u can be represented as 8-bit vectors, as can the 8-bit outputs, so as to represent f with a BDD. This permits caching to be applied, which reduces the time needed to compute a best transformer to approximately one day for each 8-bit operation. The approach does not scale to wider words nor to basic blocks.

Automatic abstraction has only recently become a practical proposition, due to the emergence of robust decision procedures and efficient quantifier elimination techniques. Our own work (cp. Chap. 5.2 and [31]) shows how bit-blasting and quantifier elimination can be applied to synthesize best transformers for bit-vectors. This work was inspired by that of Monniaux [167, 169] on automatic abstraction of piecewise linear programs, who showed that if the concrete operations are specified as piecewise linear systems, then it is possible to derive transfer functions for blocks. The key differences between our own work on quantifier-based abstraction and that of Monniaux were recently summarized by Thakur and Reps [228, Sect. 7]:

> "*Whereas Monniaux's method performs abstraction and then quantifier elimination, Brauer and King's method performs quantifier elimination on the concrete specification and then abstraction.*"

Further, although his technique is applicable to general linear template constraints, and thus extends beyond octagons, it is unclear how to express some operations (such as bit-wise logical-and) in terms of linear constraints. Universal quantification also appears in other work on inferring linear template constraints [125]. There, the authors apply Farkas' lemma to transform universal quantification into existential quantification, albeit at the cost of completeness, since Farkas' lemma prevents integral reasoning. Crucially, neither Monniaux [167] nor Gulwani et al. [125] provide a way to model integer overflow and underflow.

Thakur and Reps [228, 229] have recently presented an algorithm to compute symbolic best transformers using an adoption of Stålmarck's method [216]. The key idea of their algorithm is to incrementally find semantic reductions [78] so as to converge onto a best abstraction from above. To illustrate, suppose a concrete domain C and an abstract domain D are related through a Galois connection (C, γ, α, D) and a formula φ describes a concrete value $c \in C$. An abstract value $d' \in D$ is a semantic reduction of $d \in D$ with respect to φ iff (i) $\gamma(d') \cap \varphi = \gamma(d) \cap \varphi$, and (ii) $d' \sqsubseteq d$. The key idea of Thakur and Reps is now to improve the propagation rules in Stålmarck's method to incrementally find a semantic reduction d' of d. This approach is not dissimilar to our work on refining abstractions for octagons and other classes of linear inequalities using incremental SAT/SMT solving (cp. Chap. 4.2.1). However, our algorithms for affine and polynomial abstractions proceed diametrically opposed, by systematically converging onto the least sound abstraction from below.

The question of how to construct a best abstract transformer has also been considered in the context of Markov decision processes (MDPs), for which the first abstract interpretation framework has recently been developed by Wachter and Zhang [238]. The framework affords the calculation of both lower and upper bounds on reachability probabilities, and focuses on predicate abstraction, which has had some success with large MDPs. Given a fixed set of predicates, Wachter and Zhang seek to answer the question of what is the most precise abstract program that still is a correct abstraction. More generally, the work illustrates that the question of how to compute abstract transformers is pertinent even in the probabilistic setting.

5.6.2 Modular Arithmetic

The classical approach to handling overflows is to follow the application of a transfer function with overflow and underflow checks. Variables are then considered to be unbounded for the purpose of applying the transformer, but then their sizes are considered and, if necessary, range adjustments are applied to model wrapping. This approach has, among other tools, been implemented in Astrée [83–85]. However, even though this method appears attractive for the analysis of high-level languages and assembly languages with large registers, it is infeasible for small embedded systems devices, where wraps are used by intention. In this context, Bygde et al. [54] have shown that for convex polyhedra it is also possible to revise the concretization map to reflect truncation, building upon the work of Simon and King [223]. Both techniques allow to eliminate range tests from most abstract operations.

Another strategy to modeling machine arithmetic is to deploy congruence relationships [39, 114–116, 144, 145, 171, 172, 213] where the modulo is a power of two so as to reflect wrapping within the abstract domain itself (the domain of linear congruences is also known as the *grids* domain [5]). This approach can be applied to find both, relationships between different words [171, 172] and the bits that consti-

tute words [39, 144, 145].[3] We have also combined bit-level abstractions with range analysis [39], and so have Codish et al. [71], but neither of these works addresses the problem of relational abstraction or transfer function synthesis. Unfortunately, no meaningful notion of inequality has been found for congruence relations [172, Sect. 7], unlike for linear ones, which necessitates techniques akin to case-splitting. For example, modular arithmetic can be modeled with case-splitting by introducing a propositional variable that acts as a witness to an overflow. To illustrate, consider the 8-bit comparison $x + 100 \leq 10$ [148, Sect. 6.4]. To model overflow, a witness $p \Leftrightarrow x + 100 \leq 255$ is defined, which is used to control case-selection. Selection among different cases can, in turn, be realized through two constraints defined as $p \Rightarrow (x + 100 \leq 10)$ and $(\neg p) \Rightarrow (x + 100 - 256 \leq 10)$. Case-based axiomatizations can even be used to model underflows and rounding-to-zero in IEEE-754 floating point arithmetic [167, Sect. 4.5]. These ideas are similar in spirit to decomposing a basic block into its different modes, which are then protected by guards.

A structurally simpler domain is that of non-relational arithmetical congruences, which has been introduced by Granger [114]. In his seminal work, Granger used the notation $k + m \cdot \mathbb{Z}$ to denote the set $\{k + m \cdot z \mid z \in \mathbb{Z}\}$. We prefer a slightly different, though equivalent, notation $\equiv_m k$. Intuitively, arithmetical congruences offer a simple representation for strided value sets. Balakrishnan and Reps [9] have combined this representation with intervals — a domain for which they have coined the term *strided intervals* — which has shown useful in memory access analysis [195] as it can express both, ranges and offsets. To illustrate, suppose three consecutive 4-byte integers are indexed using an indirect read. Further, assume that these values are located at addresses 1020, 1024, and 1028 in memory. Interval analysis would infer a range $[1020, 1028]$; indirect reads for all 4-byte regions starting at $1020, \ldots, 1028$ thus need to be simulated. A strided interval $4[1020, 1028]$, in turn, can be seen as the reduced product of intervals with arithmetical congruences. It expresses both, the range of indexed memory locations and the fact that only addresses which are multiples of 4 are indeed accessed. By computing $\alpha^{\boldsymbol{V}}_{\mathsf{int}}(\varphi)$ and $\alpha^{\boldsymbol{V}}_{\mathsf{a\text{-}cong}}(\varphi)$, we could compute such a representation using our methods, too. Bygde [53] has studied the application of arithmetical congruences to low-level code analysis as well. Whereas Granger [114] provided transformers for standard operations such as addition and multiplication over unbounded integers, Bygde [53, Chap. 5] considered the case of bounded integers represented as bit-vectors and defined transformers for bit-wise operations such as `XOR`. However, all transformers were derived manually.

[3] The relative precision of these two approaches has been compared by Elder et al. [96], the result being that both domains are incomparable.

5.6.3 Polynomial Relations

The past decade has seen increasing interest in the derivation of polynomial invariants, with techniques broadly falling into two classes: (i) methods that use algebraic techniques to directly manipulate polynomials and (ii) methods that model polynomial invariants in a linear setting. The work of Colón [73] is a representative of the latter, for he shows how polynomial relations of bounded degree can be derived using program transformations. Suppose a variable a is updated using the assignment $a = a + 1$. First, an auxiliary variable s is introduced to represent the non-linear term a^2. The program is then extended by replacing the assignment $a = a + 1$ with the parallel assignment $(a, s) = (a + 1, s + 2 \cdot a + 1)$, which reflects the effects of the update of a on s. Linear invariants between a and s (and possibly other variables) in the transformed program are then interpreted as polynomial invariants. The idea of using non-linear terms as additional independent variables also arises in the approach of Bagnara et al. [4], who use convex polyhedra to represent polynomial cones of bounded degree, and thereby derive polynomial inequalities. To reduce the loss of precision incurred by linearization, they introduce additional linear inequalities which are included in the polyhedra to express non-linear constraints. The idea of extending a vector of variables with non-linear terms also arises in the work of Müller-Olm and Seidl [170], who most notably consider the complexity of inferring polynomial equalities up to a fixed degree. Müller-Olm and Seidl represent an affine relation using a set of vectors that generate the space through linear combination. Extending this idea to polynomial relations, they introduce variables that represent non-linear terms, which naturally leads to the notion of polynomial hull which is not dissimilar to our closure algorithm presented in Chap. 4.2.6.

Quantifier elimination has been proposed as a technique for inferring polynomial inequalities by Kapur [135]. In his approach, invariants are templates of polynomial inequalities with undetermined coefficients. Deriving coefficients for the templates amounts to applying quantifier elimination, which can be computed using a parametric (or comprehensive) Gröbner basis construction [241]. This approach resonates with the technique proposed by Monniaux [167, 169] for inferring loop invariants. Gröbner bases also arise in techniques for calculating invariants that are based on fixed point calculation [201, 202], the main advantage of this approach being that it does not assume any a priori bound on the degree of a polynomial. Polynomial analysis has also been applied in the field of SAT-based termination analysis by Fuhs et al. [101] using term rewriting [110, 233]. Although not strictly related to our field, their work provides techniques for encoding polynomial equality and inequality constraints in propositional Boolean logic. An easier-to-implement approach is to use SMT solvers [148] for bit-vector theories. For example, Z3 [90] or Boolector [44]) provide built-in support for polynomial expressions over bit-vectors.

5.6.4 Summary-based Program Analysis

Abstracting the effect of a procedure in a summary is a key problem in interprocedural program analysis [215] since it enables the effect of a call on an abstract state to be determined without repeatedly tracing the call. The challenge posed by summaries is how they can be densely represented whilst supporting the function composition and function application. Gen/kill bit-vector problems [196] are amenable to efficient representation, though for other problems, such as that of tracking variable equalities [173], it is better not to tabular the effect of a call directly. This is because if a transformer is distributive, then the lower adjoint of a transformer uniquely determines the transformer and, perhaps surprisingly, the lower adjoint can sometimes be represented more succinctly than the transformer itself. Acceleration [112, 155, 156, 212] is attracting increasing interest as an alternative way of computing a summary of a procedure, or more exactly the loops that it contains. The idea is to track how program state changes on each loop iteration so as to compute the trajectory of these changes (in a computation that is akin to transitive closure) and hence derive, in a single step, a loop invariant that holds on all iterations of the loop. A similar idea is found in *loop leaping* [15, 150, 151, 235]. The key idea in loop leaping, also colloquially referred to as loop frogging, is to derive a loop summary, which is applied to abstract the loop.

Symbolic bounds, which are key to our transfer functions, also arise in a form of bounds analysis [203] that aspires to infer ranges on pointer and array indices in terms of the parameters of a procedure. Bounds on each program variable at each program point are formulated as linear functions of the parameters of the function, where the coefficients are themselves parametric. The analysis problem then amounts to inferring values for these coefficients. By assuming variables to be non-negative, inequalities between the symbolic bounds can be reduced to inequalities between the parametric coefficients, thereby reducing the problem to linear programming.

5.7 Discussion

In summary, this chapter has discussed the problem of automatically computing transformers, based on a multitude of algorithms to derive abstractions for intervals, octagons, and arithmetical congruences as affine and polynomial equalities (which sometimes involve minimization and maximization), and also as Boolean functions. To structure the presentation, we have roughly categorized the techniques as lifting, quantification, and interleaved abstraction, a summary of which is given in Tab. 5.4. A commonality among all techniques discussed in this chapter is that they are based on the abstraction mechanisms introduced in Chap. 4. It should be noted that the estimated cost should be interpreted relative to cheap abstractions such as $\alpha^{\boldsymbol{V}}_{\mathsf{aff}}(\varphi)$; other methods (cp. Regehr and Reid [186]) require in excess of 24 hours *per*

Table 5.4: Overview of techniques to compute transformers presented in Chap. 5

Type	**Technique**	**Chap.**	**Precision**	**Cost**
$\mathsf{Int} \xrightarrow{\mathsf{Aff}} \mathsf{Int}$	lifting	5.1.1	rel. optimal	low
$\mathsf{Oct} \xrightarrow{\mathsf{Aff}} \mathsf{Oct}$	lifting	5.1.2	medium	low
$\mathsf{Int} \xrightarrow{\mathsf{Poly}} \mathsf{Int}$	lifting	5.1.3	rel. optimal	low
$\mathsf{Oct} \xrightarrow{\mathsf{Poly}} \mathsf{Oct}$	lifting	5.1.4	medium	low
$D_1 \xrightarrow{\mathsf{Bool}} D_2$	quantification	5.2	optimal	(very) high
$\mathsf{Oct} \xrightarrow{\mathsf{Aff}} \mathsf{Oct}$	interleaved	5.3.1	optimal	high
$\mathsf{Oct} \xrightarrow{\mathsf{Poly}} \mathsf{Oct}$	interleaved	5.3.2	optimal	high
$\mathsf{A\text{-}Cong} \xrightarrow{\mathsf{Aff}} \mathsf{A\text{-}Cong}$	interleaved	5.3.3	probab.	low
$\mathsf{Int} \xrightarrow{\mathsf{Conv}} \mathsf{Int}$	non-affine	5.4.1	high	high
$\mathsf{Oct} \xrightarrow{\mathsf{Conv}} \mathsf{Oct}$	non-affine	5.4.2	medium	high
$\mathsf{Oct} \xrightarrow{\mathsf{Conv}} \mathsf{Oct}$	interleaved	5.4.3	optimal	(very) high

instruction, whereas our techniques generate transformers in the order of seconds.

Methods that lift a system of equalities to symbolic constraints are computationally cheap, but suffer from imprecision for relational domains. This is because these techniques, which have been presented in Chap. 5.1, merely exploit the syntactic structure of the equalities on input. By way of contrast, quantification-based transformers are very expensive since the key idea is to encode symbolic constraints drawn from some template domain together with the concrete semantics of a block, and then to apply quantifier elimination. The expense of quantifier elimination thus limits the quantifier-based approach to automatic abstraction to registers with small bit-widths. However, it is important to appreciate that this method yields a direct bit-level description of symbolic constraints, and can thus represent *any* relation imposed by a basic block. Further, there is no reason why extrapolation could not as well be combined with quantifier-based abstraction so as to improve tractability.

The remaining techniques aim at finding a practical middle ground between exactitude and tractability. In particular, abstraction using octagons can be interleaved with affine abstraction to describe the symbolic constants of an output octagon as an affine combination of the symbolic inputs; for the affine case, we have presented an optimality result. Polyhedral abstraction in general turns out useful if the basic block exhibits non-linear relations. The output is then specified by a number of linear inequalities so that minimization and maximization operators appear in the transformer. As we have shown, polyhedra can also be applied to the derivation of inequalities that limit symbolic constants of octagons.

6 Complete Transformers

Model checking has the attractive property that, once a specification cannot be verified, a trace illustrating a counterexample is returned, which can be inspected by the end-user of a tool. This trace can then be replayed to identify and eliminate the defect. Counterexample traces have thus been highlighted as invaluable for fixing defects [63]. In contrast, abstract interpretation for asserting safety properties (i.e., assertions) typically summarizes traces into abstract states, thereby trading the ability to distinguish traces for computational tractability. Upon encountering a violation of the specification, it is then unclear which trace led to the violation. Moreover, since the abstract state is an over-approximation of the set of actually reachable states, a warning about a property violation may be spurious, which entails that a trace leading to an erroneous concrete state may not exist at all.

Spurious Warnings in Abstract Interpretation Given a safety property that cannot be proved correct using abstract interpretation, a trace to the beginning of the program would be similarly instructive to the tool user as in model checking. However, obtaining such a trace is hard as this trace needs to be constructed by going backwards step-by-step, starting at the property violation, until the entry of the program is reached. One approach is to apply the abstract transfer functions that were used in the forward analysis in reverse [198, 199]. However, these transfer functions are over-approximate, which implies that counterexample traces computed using this approach may be spurious, too. Yet, spurious warnings are the major hindrance for the acceptance and applicability of many static analyses, except those crafted for a specific application domain [83]. Bessey et al. [24, Sect. 1] have even conjectured that unsound static analyses might be preferable over sound ones:

> "*Circa 2000, unsoundness was controversial in the research community, though it has since become almost a de facto tool bias for commercial products and many research projects.*"

Their key argument is that the number of false positives can be traded off against missed bugs, thereby delivering tools that effectively find defects rather than prove their absence, which may explain the commercial success of bug-hunting frameworks compared to tools truly dedicated to verification.

Inferring Definite Traces using Exact Transformers Rather than giving up on soundness, we propose a practical technique to find legitimate traces that reveal actual defects within the abstract interpretation framework, thereby turning sound static analyses into practical bug-finding tools. Suppose that c denotes the concrete states of a program (in the collecting semantics) at a program location associated with an assertion (an invariant) ψ. With an abstraction d of c, a static analyzer then emits a warning iff the intersection of $\neg\psi$ and the concretization $\gamma(d)$ of d is non-empty. The emitted warning, in turn, is spurious iff the intersection of $\neg\psi$ and c is empty. Yet, obtaining a proof that a warning is indeed a spurious one is difficult, as this task can be seen as making abstract interpretations complete [105]. Our approach is different. We use the results of an over-approximate forward analysis to determine potential property violations. Then, starting from the error state, we perform backward analysis based on *complete* transformers to build up a trace from the violation of the property to the beginning of the program.

Completeness in Abstract Interpretation Completeness can be seen as the dual of soundness. Given concrete domain $(C, \sqsubseteq_C)$ with a transformer $g : C \to C$, soundness of its abstract counterpart $f : D \to D$ entails that g is described by f. Intuitively, soundness means that a loss in precision may occur in either of two ways:

- Given $c \in C$ and its optimal abstraction $d \in D$, we often have $c \sqsubset_C \gamma(d)$.
- Given $c \in C$ and its optimal abstraction $d \in D$, the application of f may incur a loss in precision. Thus, even if $c = \gamma(d)$, $g(c) \sqsubset_C \gamma(f(d))$ may hold.

Completeness, in turn, entails that such a loss in precision does not occur. Since monotone functions form a complete lattice, completeness of transformers can likewise be characterized as a property relative to two abstract domains. Giacobazzi et al. [108, Sect. 1] colloquially describe the situation as follows:

> "*While soundness is the basic requirement for any abstract interpretation, completeness is instead an ideal and uncommon situation. In this case, roughly speaking, the abstract semantics is able to take full advantage of the power of the underlying abstract domain.*"

In our work, the semantic specification consists of relations among finite bit-vectors, which can be represented completely by lifting a numerical domain such as octagons to its power-set [98, 106]. The more interesting problem is the efficient yet automatic derivation of complete transformers for such power-set liftings, i.e., given $c \in C$ and $d \in D$ such that $c = \gamma(d)$, derive $f' : D \to D$ such that $g(c) = \gamma(f'(d))$.

Complete Transformers Unfortunately, deriving complete transformers is not easy, as one has to maintain completeness during abstraction. One approach for reasoning about bit-vectors is to rest the analysis directly on the domain of Boolean formulae as we have proposed in [35], which further squares with the fact that such formulae form a Heyting domain, and thus support the computation of weakest preconditions (recall Chap. 3.1.3). As an alternative, we propose to express complete transformers as guarded updates and extend the abstractions introduced so far using a technique akin to disjunctive completion. Although this may seem similar to the problems and algorithms described in Chap. 4, the drive for completeness requires techniques different from those that merely aspire to preserve soundness. The key contribution of this chapter is an algorithm to compute such complete (or under-approximate) transformers, which then allow us to automatically provide legitimate counterexample traces or remove spurious warnings using abstract interpretation.

Outline In what follows, we briefly sketch the design of a backward analysis that derives paths to the entry of a program in Chap. 6.1. Then, we provide a worked example that explains the generation of complete abstractions in Chap. 6.2. Afterwards, Chap. 6.3 studies the algorithm formally and provides correctness results. This chapter then concludes with experimental results in Chap. 6.4, a survey of related work in Chap. 6.5, and a discussion in Chap. 6.6.

6.1 Backward Analysis for Counterexamples

Deriving a counterexample merely amounts to the arduous task of unrolling a program in reverse — starting from an erroneous state — so as to eventually obtain a path that reaches the entry to the program. During backward analysis, the states obtained can then be intersected with states computed using forward abstract interpretation to guide the search [198, Sect. 2.4]. It is well-known that finite paths form viable counterexamples for safety properties in LTL [63, p. 212], and are thus sufficient to refute invariants. Such a path can simply be computed by repeated application of backward transformers — a standard technique given that such transformers are available — and computing the meet of states derived using backward analysis with those generated using a preceding forward analysis. We thus refrain from discussing the details here and refer the reader to [35, 198, 199].

For a backward analysis that aims to provide definitive counterexamples, complete (or under-approximate) backward transformers are indeed the distinguished aspect of our work: they allow static analyzers to compute weakest preconditions within the abstract interpretation framework [108]. Weakest preconditions, in turn, can be used to characterize spurious counterexamples (cp. [105]). The remainder of this chapter thus focusses on the computation of complete transformers.

6.2 Worked Example

The ethos of our strategy for computing complete (or under-approximate) transformers is to generate guards and updates in parallel. Guards serve to describe ranges of variables, whereas the update prescribes how an input (resp. output) of a block is transformed into an output (resp. input for backward analysis). Abstraction is then applied and interleaved with a check for completeness of the intermediate result. If complete, the intermediate result forms part of the output; otherwise, the abstraction is extended disjunctively, which can be seen as a form of disjunctive completion. We illustrate the steps of this approach by means of an example.

6.2.1 Deriving Complete Abstractions

Suppose φ encodes `ADD R0 R1`, hence $\boldsymbol{V} = \boldsymbol{V}_{\mathsf{in}} \cup \boldsymbol{V}_{\mathsf{out}}$ with $\boldsymbol{V}_{\mathsf{in}} = \{\boldsymbol{r0}, \boldsymbol{r1}\}$ and $\boldsymbol{V}_{\mathsf{out}} = \{\boldsymbol{r0}\}$. However, assume that, unlike before, φ is not equipped with mode constraints, and thus $\alpha^{\boldsymbol{V}}_{\mathsf{aff}}(\varphi) = \top_{\mathsf{aff}}$; likewise, $\alpha^{\boldsymbol{V}}_{\mathsf{oct}}(\varphi) = \top_{\mathsf{oct}}$. For the sake of presentation, assume that we aim to compute a complete disjunctive characterization of φ using guards expressed as octagons and input-output relations expressed as affine equalities. Let $\mathcal{T}$ denote the set of all disjuncts in the combined update, i.e., initially $\mathcal{T} = \{(\bot_{\mathsf{oct}}, \bot_{\mathsf{aff}})\}$. Further, put $\xi = \varphi \wedge \neg\bot_{\mathsf{oct}}$, i.e., initially we have $\xi = \varphi$. Passing ξ to a solver provides a model $\mathbf{m}_1$ of ξ defined as:

$$\mathbf{m}_1 \quad = \quad \{ \quad \langle\!\langle \boldsymbol{r0} \rangle\!\rangle = 0 \quad \wedge \quad \langle\!\langle \boldsymbol{r1} \rangle\!\rangle = 0 \quad \wedge \quad \langle\!\langle \boldsymbol{r0}' \rangle\!\rangle = 0 \quad \}$$

By representing $\mathbf{m}_1$ as an octagon on the inputs, $\mathbf{m}_1$ defines a constraint $g_1 = (\langle\!\langle \boldsymbol{r0} \rangle\!\rangle = 0 \wedge \langle\!\langle \boldsymbol{r1} \rangle\!\rangle = 0)$. Then, an affine equality is computed for those inputs of φ that satisfy g_1, which gives $u_1 = (\langle\!\langle \boldsymbol{r0} \rangle\!\rangle = 0 \wedge \langle\!\langle \boldsymbol{r1} \rangle\!\rangle = 0 \wedge \langle\!\langle \boldsymbol{r0}' \rangle\!\rangle = 0)$. Due to abstraction, u_1 is sound for inputs of φ that satisfy g_1. Yet, u_1 is not necessarily complete. We commence by verifying completeness of u_1 by testing $\neg\varphi \wedge g_1 \wedge u_1$ for *unsatisfiability*; since unsatisfiable, u_1 is complete for $\varphi \wedge g_1$. Further, g_1 is the most general description of values that satisfy u_1: No $g_1' \in \mathsf{Oct}$ with $g_1 \sqsubset_{\mathsf{oct}} g_1'$ exists so that u_1 is sound and complete for $\varphi \wedge g_1'$. We thus put $\mathcal{T} = \{(g_1, u_1)\}$.

6.2.2 Extending Complete Abstractions

Thus far, $\mathcal{T}$ describes a single model $c = (0, 0, 0) \in \mathbb{Z}^3$ such that $c \models \varphi$. Our express aim is to extend $\mathcal{T}$ so that it covers a larger set $\{c_1, \ldots, c_n\} \in \wp(\mathbb{Z}^3)$, each of which satisfies $c_i \models \varphi$. The next step in deriving a complete abstraction of φ thus consists of extending $\mathcal{T}$. To do so, we define $\xi' = \xi \wedge \neg g_1$ and pass ξ' to a solver, which gives:

$$\mathbf{m}_2 \quad = \quad \{ \quad \langle\!\langle \boldsymbol{r0} \rangle\!\rangle = 2 \quad \wedge \quad \langle\!\langle \boldsymbol{r1} \rangle\!\rangle = 2 \quad \wedge \quad \langle\!\langle \boldsymbol{r0}' \rangle\!\rangle = 4 \quad \}$$

Joining g_1 with the octagon ($\langle\!\langle \boldsymbol{r0} \rangle\!\rangle = 2 \wedge \langle\!\langle \boldsymbol{r1} \rangle\!\rangle = 2$) yields $g_2 = (0 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq 2 \wedge 0 \leq \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 2)$, and we compute an affine abstraction of $\varphi \wedge g_2$, which gives:

$$u_2 \quad = \quad \alpha^{\boldsymbol{V}}_{\mathsf{aff}}(\varphi) \quad = \quad (\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle)$$

Again, testing $\neg\varphi \wedge g_2 \wedge u_2$ reveals unsatisfiability, and thus, completeness. We generalize g_2, i.e., we relax g_2 towards a more general octagon $g_2' \in \mathsf{Oct}$ with $g_2 \sqsubseteq_{\mathsf{oct}} g_2'$ for which u_2 still is a sound as well as complete abstraction. Technically, generalization can be implemented using dichotomic search, as in $\alpha^{\boldsymbol{V}}_{\mathsf{int}}(\varphi)$ or $\alpha^{\boldsymbol{V}}_{\mathsf{oct}}(\varphi)$. Of course, soundness of u_2 can be tested for $\varphi \wedge g_2'$ as in Chap. 4.2.5, i.e., by testing $\varphi \wedge g_2' \wedge \neg u_2$ for unsatisfiability. This procedure gives

$$g_2' \quad = \quad (-128 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 127)$$

as expected. Indeed, g_2' is the most general description of $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle$ and $\langle\!\langle \boldsymbol{r1} \rangle\!\rangle$ for which u_2 is sound and complete. Therefore, we replace (g_1, u_1) in $\mathcal{T}$ by (g_2', u_2).

6.2.3 Disjunctive Extensions

In the third iteration, we put $\xi'' = \varphi \wedge \neg g_2'$. Passing ξ'' to a solver yields:

$$\mathbf{m}_3 \quad = \quad \{ \quad \langle\!\langle \boldsymbol{r0} \rangle\!\rangle = 127 \quad \wedge \quad \langle\!\langle \boldsymbol{r1} \rangle\!\rangle = 127 \quad \wedge \quad \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = -2 \quad \}$$

Representing $\mathbf{m}_3$ as an octagon, which is joined with g_2 to give g_3, and computing the affine abstraction of $\varphi \wedge g_3$ then yields $u_3 = \top_{\mathsf{aff}}$. Clearly, u_3 is sound. Yet, it is incomplete, which is confirmed by the SAT solver through satisfiability of $\neg\varphi \wedge g_3 \wedge u_3$. We thus extend $\mathcal{T}$ to give $\mathcal{T}'$, representing the following transformer:

$$\mathcal{T}' \quad = \quad \left\{ \begin{array}{lcl} (-128 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 127) & \Rightarrow & (\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle) \\ (\langle\!\langle \boldsymbol{r0} \rangle\!\rangle = 127 \wedge \langle\!\langle \boldsymbol{r1} \rangle\!\rangle = 127) & \Rightarrow & (\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = -2) \end{array} \right\}$$

In the fourth iteration, we define $\xi''' = \varphi \wedge \neg g_2 \wedge \neg g_3$ and obtain a model:

$$\mathbf{m}_4 \quad = \quad \{ \quad \langle\!\langle \boldsymbol{r0} \rangle\!\rangle = -128 \quad \wedge \quad \langle\!\langle \boldsymbol{r1} \rangle\!\rangle = -1 \quad \wedge \quad \langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = 127 \quad \}$$

Proceeding much like before, this model cannot be joined with an element in $\mathcal{T}'$ without sacrificing completeness. We thus obtain:

$$\mathcal{T}'' \quad = \quad \left\{ \begin{array}{lcl} (-128 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 127) & \Rightarrow & (\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle) \\ (\langle\!\langle \boldsymbol{r0} \rangle\!\rangle = 127 \wedge \langle\!\langle \boldsymbol{r1} \rangle\!\rangle = 127) & \Rightarrow & (\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = -2) \\ (\langle\!\langle \boldsymbol{r0} \rangle\!\rangle = -128 \wedge \langle\!\langle \boldsymbol{r1} \rangle\!\rangle = -1) & \Rightarrow & (\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = 127) \end{array} \right\}$$

Using two more iterations, we obtain models that can be used to generalize ($\langle\!\langle \boldsymbol{r0} \rangle\!\rangle = 127 \wedge \langle\!\langle \boldsymbol{r1} \rangle\!\rangle = 127$) and ($\langle\!\langle \boldsymbol{r0} \rangle\!\rangle = -128 \wedge \langle\!\langle \boldsymbol{r1} \rangle\!\rangle = -1$). We thus compute the output

$$\left\{ \begin{array}{lcl} (-128 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 127) & \Rightarrow & (\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle) \\ (-256 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq -129) & \Rightarrow & (\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = 256 + \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle) \\ (128 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle \leq 254) & \Rightarrow & (\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle = -256 + \langle\!\langle \boldsymbol{r0} \rangle\!\rangle + \langle\!\langle \boldsymbol{r1} \rangle\!\rangle) \end{array} \right\}$$

together with proofs that each transformer u_i is a complete characterization of $\varphi \wedge g_i$. Since the guards describe all feasible inputs of φ, we obtain a completeness result for φ as a whole. Of course, the technique is not limited exclusively to the combination of octagons with affine equalities in forward direction, but it can also be applied in reverse direction to a wide combination of domains.

Yet, it is important to note that the above transformer, which can directly be translated into one that specifies $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle$ in terms of $\langle\!\langle \boldsymbol{r1} \rangle\!\rangle$ and $\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle$ (resp. $\langle\!\langle \boldsymbol{r1} \rangle\!\rangle$ in terms of $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle$ and $\langle\!\langle \boldsymbol{r0'} \rangle\!\rangle$), does not prescribe an exact value $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle$ (resp. $\langle\!\langle \boldsymbol{r1} \rangle\!\rangle$). Even though the transformer is complete, it is non-invertible, hence the need to intersect the states obtained using backward analysis with those computed using forward analysis.

6.3 Formalization

The procedure sketched in the worked example yields a disjunctive, sound, and complete characterization of $\varphi \in \wp(\wp(\boldsymbol{V}))$. We state prerequisites and formalize the algorithm before studying correctness and completeness.

6.3.1 Algorithm

Algorithm 14 presents a procedure $\alpha^{\boldsymbol{V}}_{\mathsf{exact}}(\varphi)$ that converges onto an exact representation of $\varphi \in \wp(\wp(\boldsymbol{V}))$ from below.

Prerequisites The procedure is parameterized by two abstract domains: $(G, \sqsubseteq_G)$ for the guards and $(U, \sqsubseteq_U)$ for the updates. For both domains, we assume Galois connections $(\wp(\mathbb{Z}^n), \gamma_G, \alpha_G, G)$ and $(\wp(\mathbb{Z}^n), \gamma_U, \alpha_U, U)$. Further, we require that for each $c \in \mathbb{Z}^n$, there exists $g \in G$ (resp. $u \in U$) such that $c = \gamma_G(g)$ (resp. $c = \gamma_U(u)$).[1] Then, the disjunctive completion of either G or U is complete w.r.t. to the base semantics expressed using Boolean formulae (cp. [106, Sect. 3.1] and [98]).

Proposition 6.1. *Let $c_1, \ldots, c_m \in \mathbb{Z}^n$ and let $(L, \sqsubseteq_L)$ be the power-set of a finite complete lattice as defined above. Since there exist $l_1, \ldots, l_m \in L$ such that $c_i = \gamma_L(l_i)$ for all $i \in 1, \ldots m$, we have $\{c_1, \ldots, c_m\} = \gamma_L(\bigvee_{i=1}^m l_i) = \bigcup_{i=1}^m \gamma_L(l_i)$.*

Of course, the reduced product of G and U is complete w.r.t. Boolean formulae, too (or equivalently, $\wp(\mathbb{Z}^n)$). In the algorithm, we represent guarded updates using the set $\mathcal{T} \in \wp(G \times U)$. Then, if $\mathcal{T} = \{(g_1, u_1), \ldots, (g_m, u_m) \mid g_i \in G, u_i \in U\}$, the concretization of $\mathcal{T}$ is interpreted as $\bigcup_{i=1}^m (\gamma_G(g_i) \cap \gamma_U(u_i))$.

[1] All relational domains and also conjunctions of the non-relational ones studied in this dissertation satisfy this requirement. This is not always so, consider, e.g., the abstract domain of signs.

Iteration The key idea of $\alpha^{\boldsymbol{V}}_{\mathsf{exact}}(\varphi)$ is to find a model $\mathbf{m}$ of φ that is not covered by the intermediate result $\mathcal{T}$ (line 3), i.e., $\mathbf{m} \notin \bigcup_{(g,u)\in\mathcal{T}}(\gamma_G(g)\cap\gamma_U(u))$. If such a model $\mathbf{m}$ is found, then the procedure attempts to extend an existing guarded update in $\mathcal{T}$ without losing precision. It does so by iterating over all elements $(g,u) \in \mathcal{T}$, trying to find one that can be generalized towards $(g_{\mathsf{new}}, u_{\mathsf{new}})$ with $g \sqsubset_G g_{\mathsf{new}}$ and $u \sqsubset_U u_{\mathsf{new}}$ without sacrificing completeness (lines 5–14), i.e., $g_{\mathsf{new}} \wedge u_{\mathsf{new}} \models \varphi$.

The completeness criterion of $(g_{\mathsf{new}}, u_{\mathsf{new}})$ with respect to φ is straightforwardly specified as $\forall \boldsymbol{V} : (g_{\mathsf{new}} \wedge u_{\mathsf{new}}) \Rightarrow \varphi$ as in line 8. However, putting $(g_{\mathsf{new}} \wedge u_{\mathsf{new}}) \Rightarrow \varphi$ into CNF introduces fresh, existentially quantified variables T, which gives an equisatisfiable formula ψ such that $(g_{\mathsf{new}} \wedge u_{\mathsf{new}}) \Rightarrow \varphi \equiv \exists T : \psi$. Rather than testing the formula $\forall \boldsymbol{V} : \exists T : \psi$ for satisfiability, we observe the following equivalence:

$$\begin{aligned}
& \forall \boldsymbol{V} : (g_{\mathsf{new}} \wedge u_{\mathsf{new}}) \Rightarrow \varphi \\
\Leftrightarrow\ & \forall \boldsymbol{V} : \neg(g_{\mathsf{new}} \wedge u_{\mathsf{new}}) \vee \varphi \\
\Leftrightarrow\ & \forall \boldsymbol{V} : (\neg g_{\mathsf{new}} \vee \neg u_{\mathsf{new}}) \vee \varphi \\
\Leftrightarrow\ & \neg\exists \boldsymbol{V} : \neg((\neg g_{\mathsf{new}} \vee \neg u_{\mathsf{new}}) \vee \varphi) \\
\Leftrightarrow\ & \neg\exists \boldsymbol{V} : g_{\mathsf{new}} \wedge u_{\mathsf{new}} \wedge \neg\varphi
\end{aligned}$$

To determine satisfaction of $\forall \boldsymbol{V} : (g_{\mathsf{new}} \wedge u_{\mathsf{new}}) \Rightarrow \varphi$, it thus suffices to put $g_{\mathsf{new}} \wedge u_{\mathsf{new}} \wedge \neg\varphi$ into CNF, which gives an equisatisfiable formula ψ', and test $\exists \boldsymbol{V} : \exists T' : \psi'$ for *unsatisfiability*. Soundness of the input-output relation u_{new} subject to g_{new} comes for free, as it is derived using abstraction, which entails $g_{\mathsf{new}} \wedge \varphi \models u_{\mathsf{new}}$. If such a pair $(g_{\mathsf{new}}, u_{\mathsf{new}})$ that satisfies the completeness criterion is found, the algorithm attempts to generalize it, i.e., find a guard $g_{\mathsf{gen}} \in G$ with $g_{\mathsf{new}} \sqsubset_G g_{\mathsf{gen}}$ such that completeness is preserved, i.e., $\neg\varphi \wedge g_{\mathsf{gen}} \wedge u_{\mathsf{new}}$ is unsatisfiable. Otherwise, if the current model $\mathbf{m}$ cannot be combined with an element of $\mathcal{T}$ to give a complete approximation of φ, we extend $\mathcal{T}$ disjunctively (lines 15–17). Observe that for domains $(G, \sqsubseteq_G)$ that express ranges, the procedure `generalize` can be implemented exactly using dichotomic search; this means that for a guard g_{new} and an update u_{new}, the most general guard g_{gen} such that $g_{\mathsf{new}} \sqsubseteq_G g_{\mathsf{gen}}$ and $g_{\mathsf{gen}} \wedge u_{\mathsf{new}} \models \varphi$ is found.

Reprise and Reflection It is noteworthy that $\alpha^{\boldsymbol{V}}_{\mathsf{exact}}(\varphi)$ does not require a formula $\varphi \in \wp(\wp(\boldsymbol{V}))$ on input that is equipped with mode constraints. By way of contrast, the abstractions introduced in Chap. 4 require these encodings of mode combinations. This requirement is finessed by computing exact guards and updates in parallel, rather than separately, introducing a fresh disjunction once precision is lost. This strategy thus entails that an exact representation of φ is eventually derived, the drawback being that $\alpha^{\boldsymbol{V}}_{\mathsf{exact}}(\varphi)$ may be significantly more expensive to compute. Observe too that the procedure is not limited to forward analysis, as it provides formulae that represent direct input-output relations between bit-vectors with the same precision as φ does, yet in a human-readable way that does not require the repeated application of a SAT solver to compute the outputs.

Algorithm 14 $\alpha^{\boldsymbol{V}}_{\mathsf{exact}}$

1: $\mathcal{T} \leftarrow \{(\bot_G, \bot_U)\}$
2: $\xi \leftarrow \varphi$
3: **while** ξ is satisfiable with model **m** **do**
4: $\quad \mathsf{success} \leftarrow \mathsf{false}$
5: $\quad$ **for** each $(g, u) \in \mathcal{T}$ **do**
6: $\quad\quad g_{\mathsf{new}} \leftarrow g \sqcup_G \alpha^{\boldsymbol{V}}_G(\mathbf{m})$
7: $\quad\quad u_{\mathsf{new}} \leftarrow \alpha^{\boldsymbol{V}}_U(\varphi \wedge g_{\mathsf{new}})$
$\quad\quad$ {check for completeness}
8: $\quad\quad$ **if** $\neg\varphi \wedge g_{\mathsf{new}} \wedge u_{\mathsf{new}}$ are unsatisfiable **then**
9: $\quad\quad\quad g_{\mathsf{gen}} \leftarrow \mathtt{generalize}(\varphi, g_{\mathsf{new}}, u_{\mathsf{new}})$
10: $\quad\quad\quad \mathcal{T} \leftarrow \mathcal{T} \setminus \{(g, u)\} \cup \{(g_{\mathsf{gen}}, u_{\mathsf{new}})\}$
11: $\quad\quad\quad \mathsf{success} \leftarrow \mathsf{true}$
12: $\quad\quad\quad$ **break**
13: $\quad\quad$ **end if**
14: $\quad$ **end for**
15: $\quad$ **if** $\neg\mathsf{success}$ **then**
16: $\quad\quad \mathcal{T} \leftarrow \mathcal{T} \cup \{(\alpha^{\boldsymbol{V}}_G(\mathbf{m}), \alpha^{\boldsymbol{V}}_U(\mathbf{m}))\}$
17: $\quad$ **end if**
18: $\quad \xi \leftarrow \varphi \wedge \neg(\bigvee_{(g,u)\in\mathcal{T}} g)$
19: **end while**
20: **return** $\mathcal{T}$

Optimization Procedure $\alpha^{\boldsymbol{V}}_{\mathsf{exact}}(\varphi)$ computes a complete abstractions that cover all feasible inputs. This approach may lead to a large number of disjuncts (i.e., pairs $(g, u) \in \mathcal{T}$), which also depends on the abstract domains used within $\alpha^{\boldsymbol{V}}_{\mathsf{exact}}(\varphi)$. When the procedure is implemented within abstract interpretation frameworks to infer definitive counterexample traces, deriving a complete transformer is necessary only for reachable states, an over-approximation of which has already been computed using forward fixed-point iteration. A straightforward optimization that does not affect soundness is thus to restrict $\alpha^{\boldsymbol{V}}_{\mathsf{exact}}(\varphi)$ to the sub-range of reachable states. Given states s and s' on input and output of a block b encoded by φ, it is then sufficient to compute $\alpha^{\boldsymbol{V}}_{\mathsf{exact}}(\varphi \wedge s \wedge s')$. This approach is not dissimilar to the technique of Rival [198, Sect. 2.4], who observed that weakest preconditions are often not sufficiently precise. To finesse this problem, he designed a backward transformer as a monotone transfer function of two arguments: (1) an invariant to refine, and (2) and invariant to propagate backwards. Even though not necessary for correctness, there is also no reason why φ could not be augmented with mode constraints so as to simplify the formula and the derivation of a complete abstraction.

6.3.2 Soundness and Completeness

Here, we argue about correctness of $\alpha^{\boldsymbol{V}}_{\mathsf{exact}}(\varphi)$, showing that it is sound as well as complete with respect to φ.

Theorem 6.1. *Let $\varphi \in \wp(\wp(\boldsymbol{V}))$. Then, $\alpha^{\boldsymbol{V}}_{\mathsf{exact}}(\varphi) \models \varphi$.*

Proof. To prove this theorem, we show that $\bigvee_{(g,u)\in\mathcal{T}}(\gamma_G(g) \cap \gamma_U(u)) \models \varphi$ is an invariant of Alg. 14. Initially, $\mathcal{T} = \{(\bot_G, \bot_U)\}$, hence, $\bigvee_{(g,u)\in\mathcal{T}}(\gamma_G(g) \cap \gamma_U(u)) \models \varphi$. Within the loop, ξ is satisfiable, hence there exists $c \in \mathbb{Z}^n$ such that $c \models \xi$ and $c \notin \bigcup_{(g,t)\in\mathcal{T}} \gamma_G(g)$, corresponding to the model $\mathbf{m}$ in Alg. 14. For $(g, u) \in \mathcal{T}$, let $g_{\mathsf{new}} = g \sqcup_G \alpha_G(c)$ and $u_{\mathsf{new}} = u \sqcup_U \alpha_U(c)$, and we consider two cases separately:

- $\gamma_G(g_{\mathsf{new}}) \cap \gamma_U(u_{\mathsf{new}}) \models \varphi$. As argued before, $\neg\varphi \wedge g_{\mathsf{new}} \wedge u_{\mathsf{new}}$ is unsatisfiable, and `generalize` preserves this property, whence $\gamma_G(g_{\mathsf{gen}}) \cap \gamma_U(u_{\mathsf{new}}) \models \varphi$. Since $\bigcup_{(g,u)\in\mathcal{T}}(\gamma_G(g) \cap \gamma_U(u)) \models \varphi$, we have $(\bigcup_{(g,u)\in\mathcal{T}}(\gamma_G(g) \cap \gamma_U(u))) \cup (\gamma_G(g_{\mathsf{gen}}) \cap \gamma_U(u_{\mathsf{new}})) \models \varphi$, as desired.

- $\gamma_G(g_{\mathsf{new}}) \cap \gamma_U(u_{\mathsf{new}}) \not\models \varphi$. With Prop. 6.1, we have $\gamma_G(c) \cap \gamma_U(c) = c$ and also $c \models \varphi$, whence $\bigvee_{(g,u)\in\mathcal{T}}(\gamma_G(g) \cap \gamma_U(u)) \cup c \models \varphi$, as desired.

By combining both cases, we obtain a proof of our claim. □

An immediate consequence of the loop invariant as constructed in the proof of Thm. 6.1 is that $\alpha^{\boldsymbol{V}}_{\mathsf{exact}}(\varphi)$ can be interrupted prematurely without sacrificing validity of Thm. 6.1. This is significant as computing $\alpha^{\boldsymbol{V}}_{\mathsf{exact}}(\varphi)$ exactly may be expensive, while an transformer $\mathcal{T}'$ such that $\mathcal{T}' \models \mathcal{T}$ may be sufficient to generate a counterexample, albeit not to show spuriousness of a warning. We further observe:

Theorem 6.2. *Let $\varphi \in \wp(\wp(\boldsymbol{V}))$. Then, $\varphi \models \alpha^{\boldsymbol{V}}_{\mathsf{exact}}(\varphi)$.*

Proof. The algorithm terminates once $\varphi \wedge \neg(\bigvee_{(g,u)\in\mathcal{T}} g)$ is unsatisfiable. Let $c \in \mathbb{Z}^n$ such that $c \models \varphi$. Then, there exists $(g, u) \in \mathcal{T}$ such that $c \in \gamma_G(g)$. Further, we have $u = \alpha_U(\varphi \wedge g)$ by construction, whence $c \in \gamma_U(u)$. It follows that $c \in \gamma_G(g) \cap \gamma_U(u)$, whence $\varphi \models \alpha^{\boldsymbol{V}}_{\mathsf{exact}}(\varphi)$. □

Corollary 6.1. *Let $\varphi \in \wp(\wp(\boldsymbol{V}))$. Then, $\alpha^{\boldsymbol{V}}_{\mathsf{exact}}(\varphi)$ is sound and complete w.r.t. φ.*

To conclude, observe that $\alpha^{\boldsymbol{V}}_{\mathsf{exact}}(\varphi)$ does not necessarily provide minimal or unique abstractions. As an example, suppose $\varphi \in \wp(\wp(\{\boldsymbol{x}, \boldsymbol{y}\}))$ has satisfying assignments $\{(-128, -1), \ldots, (-1, -1), (0, 0), (1, 1), (127, 1)\}$. Then, depending on the order in

which models are provided, $\alpha^{\boldsymbol{V}}_{\mathsf{exact}}(\varphi)$ based on intervals and affine equalities could produce different yet equivalent abstractions, e.g.:

$$(1)\quad \begin{cases} (-128 \leq \langle\!\langle \boldsymbol{x} \rangle\!\rangle \leq -1) & \Rightarrow & (\langle\!\langle \boldsymbol{y} \rangle\!\rangle = -1) \\ (0 \leq \langle\!\langle \boldsymbol{x} \rangle\!\rangle \leq 0) & \Rightarrow & (\langle\!\langle \boldsymbol{y} \rangle\!\rangle = 0) \\ (1 \leq \langle\!\langle \boldsymbol{x} \rangle\!\rangle \leq 127) & \Rightarrow & (\langle\!\langle \boldsymbol{y} \rangle\!\rangle = 1) \end{cases}$$

$$(2)\quad \begin{cases} (-128 \leq \langle\!\langle \boldsymbol{x} \rangle\!\rangle \leq -2) & \Rightarrow & (\langle\!\langle \boldsymbol{y} \rangle\!\rangle = -1) \\ (-1 \leq \langle\!\langle \boldsymbol{x} \rangle\!\rangle \leq 1) & \Rightarrow & (\langle\!\langle \boldsymbol{y} \rangle\!\rangle = \langle\!\langle \boldsymbol{x} \rangle\!\rangle) \\ (2 \leq \langle\!\langle \boldsymbol{x} \rangle\!\rangle \leq 127) & \Rightarrow & (\langle\!\langle \boldsymbol{y} \rangle\!\rangle = 1) \end{cases}$$

6.4 Experiments

Our implementation of $\alpha^{\boldsymbol{V}}_{\mathsf{exact}}(\varphi)$ is written in C++ on top of Z3. The performance of the procedure depends on (1) the abstract domains that are plugged into $\alpha^{\boldsymbol{V}}_{\mathsf{exact}}(\varphi)$ and (2) the relations described by φ. This is because the expressiveness of the respective abstract domain of course affects the number of disjuncts in the resulting abstraction, and thus the number of iterations required to compute it. For `ADD R0 R1`, the overhead for $\alpha^{\boldsymbol{V}}_{\mathsf{exact}}(\varphi)$ based on octagons and affine equalities compared to separate computation of $\alpha^{\boldsymbol{V}}_{\mathsf{oct}}(\varphi)$ and $\alpha^{\boldsymbol{V}}_{\mathsf{aff}}(\varphi)$ is imperceivable, likewise for other arithmetic operations such as `SUB R0 R1` or `INC R0` and combinations thereof.

6.4.1 Effects of Domain Combinations

By way of contrast, complete abstraction for `LSL R0` based on intervals and affine equalities requires approximately 1 minute to complete for the 8-bit case, whereas the combination of arithmetical congruences and affine equalities terminates within 0.1 seconds. From our experience, the most effective strategy on average is to combine several domains, e.g., arithmetical congruences and octagons, to form $(G, \sqsubseteq_G)$ with a domain that captures equalities. In this case, the output of $\alpha^{\boldsymbol{V}}_{\mathsf{exact}}(\varphi)$ for `ASR R0` is:

$$\begin{array}{lcl} ((\langle \boldsymbol{r0} \rangle \equiv_2 0) \wedge (0 \leq \langle \boldsymbol{r0} \rangle \leq 254)) & \Rightarrow & (\langle \boldsymbol{r0'} \rangle = \frac{1}{2} \cdot \langle \boldsymbol{r0} \rangle) \\ ((\langle \boldsymbol{r0} \rangle \equiv_2 1) \wedge (1 \leq \langle \boldsymbol{r0} \rangle \leq 255)) & \Rightarrow & (\langle \boldsymbol{r0'} \rangle = \frac{1}{2} \cdot (\langle \boldsymbol{r0} \rangle - 1)) \end{array}$$

Likewise, expressing guards as value sets gives the equivalent abstraction:

$$\begin{array}{lcl} (\langle \boldsymbol{r0} \rangle \in \{0, 2, \ldots, 252, 254\}) & \Rightarrow & (\langle \boldsymbol{r0'} \rangle = \frac{1}{2} \cdot \langle \boldsymbol{r0} \rangle) \\ (\langle \boldsymbol{r0} \rangle \in \{1, 3, \ldots, 253, 255\}) & \Rightarrow & (\langle \boldsymbol{r0'} \rangle = \frac{1}{2} \cdot (\langle \boldsymbol{r0} \rangle - 1)) \end{array}$$

Abstraction based on intervals and affine equalities, in turn, requires 12 seconds and eventually yields a representation that can be simplified into 128 different disjuncts.

This is because the relation described by `ASR R0` is affine only for adjacent inputs:

$$\begin{array}{lcl} (0 \leq \langle \boldsymbol{r0} \rangle \leq 1) & \Rightarrow & (\langle \boldsymbol{r0'} \rangle = 0) \\ (2 \leq \langle \boldsymbol{r0} \rangle \leq 3) & \Rightarrow & (\langle \boldsymbol{r0'} \rangle = 1) \\ \ldots & \Rightarrow & \ldots \\ (254 \leq \langle \boldsymbol{r0} \rangle \leq 255) & \Rightarrow & (\langle \boldsymbol{r0'} \rangle = 127) \end{array}$$

This situation is close to the worst case as it prevents contiguous ranges to be summarized in a single disjunct. By combining abstractions for value sets, ranges (using intervals and octagons, depending on the number of variables) and arithmetical congruences, we were able to compute complete abstractions for either 8-bit benchmark introduced in Chap. 4.4 within less than 10 seconds.

6.4.2 Complete Extrapolation

A promising combination of techniques that may not be obvious in the first place is to pair $\alpha^{V}_{\mathsf{exact}}(\varphi)$ with extrapolation (recall Chap. 4.3). Then, complete abstractions are derived for short bit-vectors of length 4 and 5, respectively, which is cheap in terms of computational cost, and extrapolation is applied to extrude the results. As demonstrated in Chap. 4.4 and Chap. 5.5, this technique works surprisingly well for deriving sound abstractions. Results obtained for complete abstraction are promising, too. Assume that a sound abstraction a is derived for φ by extrapolating two complete abstractions a' and a'' over bit-vectors of width 4 and 5, respectively. Then, completeness of a follows from unsatisfiability of $\neg\varphi \wedge a$, which can easily be checked. For rather complex blocks such as `ABS` and `ISIGN`, we have generated complete abstractions based on extrapolation in less than 0.5 seconds, which confirms our intuition.

6.5 Related Work

A sound static analysis, which is usually expressed using the abstract interpretation framework [77], is bound to calculate an over-approximate result to elude undecidability.[2] Due to over-approximation, a safety property may not be verifiable even though it holds. In this case, the emitted warning is a so-called *false positive* [24] (also called *spurious*) which cannot a priori be distinguished from an actual defect. While an analysis with zero false positives is possible [83], it is crucial to understand the origin of each alarm in order to either refine the analysis or to fix the defect. Thus, analyzing warnings which are emitted poses two related questions: (1) Is the warning legitimate? (2) If so, how can the error state be reached in terms of

[2] Of course, problems over finite bit-vectors with finite memory are decidable. Then, over-approximation is applied to avoid state explosion, and thus, to serve tractability.

a concrete execution? The difficulty of answering the first has led to approaches that rank warnings based on the likelihood of being actual defects. Statistical classifications have been based on error correlation [147] or bayesian filtering [133]. Recent work of Lee et al. [154] clusters defects, allowing to eliminate dependent defects if a master defect is shown to be spurious. Using their technique, defects can be proven legitimate, too. However, the work of Lee et al. does not tackle the problem of determining whether a master defect indeed is spurious or legitimate.

6.5.1 Counterexamples in Model Checking

An exact answer to both questions is required in counterexample-guided abstraction refinement (CEGAR) in model checking [65]. However, deciding if a warning is legitimate is strictly easier in the context of CEGAR than in a general static analysis as the model checker produces an abstract counterexample, the validity of which then can be checked. A concrete counterexample may, for example, be inferable by replaying the trace in the concrete program [149]. If successful, the concrete trace can be used afterwards for, e.g., error localization [13]. If constructing the trace fails at a certain program point, a new predicate can be introduced to refine the abstract model [14], which then suppresses the spurious trace, and possibly other spurious traces. Finitization, as performed by our approach to counterexample generation, also appears in bounded model checking [66]. There, the state space of the system is explored in a breadth-first fashion in forward direction, up to a given depth k. By way of comparison, our approach unrolls the program back-to-front.

6.5.2 Counterexamples in Abstract Interpretation

In the context of numeric analysis, Gulavani and Rajamani [120] propose to refine a pre-analysis, which is based on a fixed-point computation with widening, by introducing predicates using so-called hints. Later, they extended their technique to combine widening with interpolants between verification conditions and the inferred state [122]. Yet, neither work is concerned with computing the backward trace but assumes that it has been inferred by a theorem prover.

For static analyses that operate on the semantics of the actual program, no model program exists in which the trace can be inferred, and backward reasoning from the warning to the program entry is required [198, 199]. Backward reasoning, in turn, amounts to computing weakest preconditions (or solving abduction). Few classes of linear constraints allow abduction [160, 161] and no single numeric domain commonly used in forward analyses is Heyting, nor is the combination of Heyting domains necessarily a Heyting domain [162]. One way out of this dilemma is to lift a non-Heyting domain to its power-set domain [143], which yields a Boolean domain. A Boolean domain is always Heyting since for each element there exists

a full complement. Boolean functions naturally form a Boolean domain. Yet, the domain suffers from tractability issues, which motivates our decision to lift standard numerical domains to their power-sets. Rival [198] sidesteps the abduction problem by calculating an A' with $A \models A'$ using the same domains as in forward analysis. To cap the over-approximation of the backward transformer, backward states are intersected with the forward invariants. Over-approximation in the backward transformer makes it unlikely that an empty state is ever observed. Then, a warning cannot be identified as a false positive. Indeed, Rival's analysis merely informs tool-users about inputs in which a counterexample might lie, thereby providing "*some support in the alarm investigation process*" [198, Sect. 1]. Contemporaneously to Rival [198], Howe et al. [130] studied backward analysis for logic programs, with pair sharing analysis as one application [158]. However, the analysis of [158] is based on a condensing domain — intuitively, such a domain allows a goal-dependent analysis to be implemented in a goal-independent way without incurring a loss in precision [158, Sect. 3] — rather than a complete one as in our approach (any complete domain is condensing, but condensing domains are complete only with respect to unification [109]). Further afield is the work of Kim et al. [139] who, after a fast but imprecise forward analysis, slice the program for a property violation before running a more expensive forward analysis based on SMT solving.

6.5.3 Completeness in Abstract Interpretation

Completeness in abstract interpretation has already been considered by Cousot and Cousot [78, Sect. 7]. However, from their work, it took more than two decades until Giacobazzi et al. [108] provided a constructive characterization of completion for the general case. Most notably, they require a Scott-continuous transformer, which is a very general assumption, and then show how the problem of making abstract interpretations complete can be reduced to minimal extensions or refinements of the domain. In essence, their technique amounts to computing the least fixed point of a characterization that involves repeatedly computing closures. Our method for $\alpha^{V}_{\mathsf{exact}}(\varphi)$ is, of course, a significantly less general result, as we require an expressive (disjunctive) domain D and use incremental SAT solving to converge onto a complete transformer, which is possible due to finiteness of the base domain of Boolean formulae. This approach can indeed be seen as minimally extending the transformer to capture certain effects of φ. Later, Giacobazzi and Quintarelli [105] showed that CEGAR indeed fits into methodology of completing abstract interpretations. As far as we are aware, our algorithm is the first that effectively computes complete transformers for bit-vectors.

6.6 Discussion

The focus of Chap. 3, Chap. 4, and Chap. 5 was over-approximation, with the aim of deriving abstractions and transformers that subsume the reachable states and transition relations of a concrete program, whilst being as precise as possible. The topic of this chapter is different, although refinements based on backward interpretation have been weaved into the forward analysis in Chap. 3.2 already. It is that of computing transformers (or abstractions) that are complete (or under-approximate) w.r.t. the base semantics, i.e., propositional Boolean formulae φ, which is achieved using a parametric procedure $\alpha^{\boldsymbol{V}}_{\mathsf{exact}}(\varphi)$. The procedure is itself parametric in two abstract domains:

- a domain $(G, \sqsubseteq_G)$ to specify ranges or sets of values, and
- the other domain $(U, \sqsubseteq_U)$ to specify equalities symbolically.

Note, however, that this limitation to two domains is inconsequential for correctness. It merely entails that transformers can often be represented compactly.

For finite domains, disjunctive completion can always be achieved, and there has been much interest in optimality, e.g., using power-set liftings [98] or closures [106, Sect. 3]. However, as our work is not concerned with domain constructions, but rather the computation of complete transformers on complete domains, we have concentrated on power-set liftings of standard domains (e.g., octagons paired with affine equalities). The drawback of complete abstract interpretation [108] is its obvious high cost for both, representing the reachable states and deriving transformers. In the worst case, this approach entails that one has to resort to a representation that is as complex as the Boolean base semantics itself. Thus, instead of attempting to perform complete abstract interpretation directly, we apply a method that performs complete backward analysis only upon encountering a potential property violation, trying to find a path to the program entry by going backward step-by-step. Ideally, there are few such warnings.

7 Conclusion

Abstract interpretation provides a methodology that guarantees sound approximations of all states reachable in any concrete execution of a program. Among other algorithmic aspects, correctness of abstract interpretation ultimately depends on correctness of transfer functions for any program statement in any abstract domain used. This is not without problems (cp. [184]) since designing and implementing transfer functions is indeed a challenging task, especially if the operations are low-level and diverse [115], as in binary analysis [9, 185, 186]. Prior to our work, the state-of-the-art in abstraction interpretation for bit-vectors was manual design of transformers for each operation in a program.

7.1 Discussion

This dissertation can be seen as a response to this unsatisfactory situation in the sense that it advocates automatic abstraction as a key component of abstract interpretation frameworks. Particularly, we have discussed a collection of techniques that eliminate the need to handcraft transformers for the widely used numerical domains of intervals, value sets, octagons, convex polyhedra, arithmetical congruences, affine equalities, and bounded polynomials altogether, based on relational encodings of the concrete semantics of instructions and basic blocks. A commonality of all techniques is that they exploit the structure of the underlying abstract domain to guide the search for a sound (and optimal) abstraction. The search is, in turn, implemented using incremental SAT solving.

In a nutshell, the desire to reason about instructions that operate on finite machine words — rather than unbounded integers — manifests itself in two notable design decisions.

Relational Semantics We model each instruction in the domain of propositional Boolean formulae. Encodings for entire blocks are then derived by relational composition of the instructions that constitute the respective block. This approach confers two significant advantages:

1. Low-level operations can straightforwardly be expressed in Boolean logic.
2. Automatic abstraction uses off-the-shelf solvers and thus directly benefits from any progress made on these solvers.

Finite Machine Arithmetic Instead of targeting wrap-around arithmetic using modular domains — which are notoriously difficult to support — we identify over- and underflow modes prior to the analysis, and then derive a transformer for each feasible mode combination. This choice leads to a formulation of transfer functions as guarded updates. Then, a guard describes a class of inputs that satisfy a mode combination, whereas an update stipulates how a class on input is transformed into a class on output.

As a first application, we have discussed the problem of value set analysis for control flow reconstruction in Chap. 3. Indeed, the runtimes of the analysis are much smaller than we expected initially, given that a SAT solver is invoked on any application of a transformer. However, for relational abstract domains such as octagons or convex polyhedra, this form of online evaluation of transformers becomes intractable, which can be seen from the runtimes presented in Chap. 4.4 and Chap. 5.5. For relational abstractions, we thus compute transformers prior to the analysis itself. Computing the output of a block then amounts to evaluating a (linear or polynomial) map, rather than invoking a decision procedure. Apart from being correct by construction in a principled way, our approach features two more interesting properties:

1. We generate symbolic best transformers, which are optimal in the sense that more descriptive abstractions do not exist (in the respective abstract domain).

2. We generate transformers for blocks rather than individual instructions. We thus obtain more precise transformers than possible by computing abstractions separately, instruction by instruction.

Computing transformers instead of designing them manually eliminates one important cause for incorrect implementations of abstract interpretation and yields optimal abstractions, too. Yet, the correctness argument comes at a price: We have to assume correctness of the underlying decision procedure, which is not always easy to establish [45]. However, we have not observed unexpected results from a solver in our experiments at all and have also cross-checked the generated abstractions against results obtained using explicit-state model checking with [MC]SQUARE [207].

A distinguished feature of our framework is that slight variations of the discussed algorithms are sufficient to generate complete (or under-approximate) abstractions rather than over-approximations. This feature makes our approach amenable to the generation of counterexample traces or the elimination of spurious warnings. Even though under-approximation integrates with abstract interpretation as smoothly as over-approximation does, comparably few known techniques intentionally use such constructions; this may be explained by the difficulty of designing descriptive under-approximate or even complete transformers. Specifically, our technique structurally depends on some form of power-set construction as the abstract domain

of choice needs to be able to capture any possible relation between data present in the concrete program (although completeness can likewise be expressed using closures [106, 108]). The technique presented in Chap. 6 does so by incrementally generalizing under-approximations so as to converge onto the exact semantics of a block from below. If a certain relation cannot be represented by generalizing the current under-approximation, a fresh guarded update, which captures the missing relation, is introduced. Our approach can thus be seen as a form of disjunctive completion [78, 98, 106, 107].

7.2 Summary

In summary, for programs that operate on finite machine words, we recommend to integrate the computation of symbolic best transformers directly into abstract interpretation frameworks, as this approach

- provides most descriptive approximations of basic blocks,
- has become tractable due to progress on automated decision procedures, and
- at the same time limits the workload of implementing abstract interpretations.

Crucially for precision, if the expressiveness of a given abstraction is not sufficient to prove correctness of a program, then it is always possible to resort to a more descriptive domain, for which abstractions can automatically be generated, too.

7.3 Future Work

Analysis of Loops The problem of computing transformers for blocks leads to the more general problem of synthesizing transformers for entire loops. Computational techniques for deriving transformers for loops are colloquially referred to as *loop leaping*, capturing the central idea of jumping over the computational obstacle presented by reaching, iterating, and stabilizing on each loop (or a nest thereof) in a program. Existing approaches to specify least inductive loop invariants, which can be applied to derive loop transformers, rely on specifications using alternating quantification that are inherently difficult to solve [135, 167, 169]. Translating these approaches into propositional Boolean logic directly is intractable for all except the smallest bit-vectors. It may therefore be interesting to evaluate whether extrapolation as in Chap. 4.3 can be combined with the generation of loop transformers. Then, a candidate for a least inductive loop invariant would be derived from a specification over, e.g., 5 bits, and soundness of the candidate could be validated a posteriori.

Termination Analysis Interestingly, synthesizing transformers using quantification is not dissimilar to the problem of inferring ranking functions for bit-vectors. Given a path π with a transition relation $r_\pi(\boldsymbol{V}_{\mathsf{in}}, \boldsymbol{V}_{\mathsf{out}})$ that relates inputs $\boldsymbol{V}_{\mathsf{in}}$ and outputs $\boldsymbol{V}_{\mathsf{out}}$, proving the existence of a ranking function amounts to solving the formula

$$\exists \boldsymbol{c} : \forall \boldsymbol{V}_{\mathsf{in}} : \forall \boldsymbol{V}_{\mathsf{out}} : r_\pi(\boldsymbol{V}_{\mathsf{in}}, \boldsymbol{V}_{\mathsf{out}}) \rightarrow (p(\boldsymbol{c}, \boldsymbol{V}_{\mathsf{out}}) < p(\boldsymbol{c}, \boldsymbol{V}_{\mathsf{in}}))$$

where p is a polynomial over bit-vectors with coefficients $\boldsymbol{c}$ [75, Thm. 2]. If intermediate variables are needed to express $r_\pi(\boldsymbol{V}_{\mathsf{in}}, \boldsymbol{V}_{\mathsf{out}})$ or $p(\boldsymbol{c}, \boldsymbol{V}_{\mathsf{out}}) < p(\boldsymbol{c}, \boldsymbol{V}_{\mathsf{in}})$, then the formula actually takes the form

$$\exists \boldsymbol{c} : \forall \boldsymbol{V}_{\mathsf{in}} : \forall \boldsymbol{V}_{\mathsf{out}} : \exists \boldsymbol{T} : \nu$$

over fresh variables $\boldsymbol{T}$ where ν is equisatisfiable to:

$$r_\pi(\boldsymbol{V}_{\mathsf{in}}, \boldsymbol{V}_{\mathsf{out}}) \rightarrow (p(\boldsymbol{c}, \boldsymbol{V}_{\mathsf{out}}) < p(\boldsymbol{c}, \boldsymbol{V}_{\mathsf{in}}))$$

This formula is structurally similar to those solved in Chap. 5.2, which begs the question whether the problem of generating ranking functions can — like that of automatic abstraction — be recast to avoid quantifier elimination altogether. In the light of contemporary research, which investigates termination in the framework of abstract interpretation [81], and the development of decision procedures that target termination [176], this research direction appears promising.

Floating Point Arithmetic To conclude, this dissertation has focussed on (signed and unsigned) integer arithmetic over finite machine words, whereas floating point numbers have not been investigated. Recent research by D'Silva et al. [93], to name one example, provides a decision procedure to reason about floating point intervals so as to finesse the problems incurred by analyzing floating point numbers at the granularity of individual bits. Such an approach could smoothly integrate with our work, and evaluating the generation of abstractions for floating point arithmetic on top of such a decision procedure appears promising. Yet, small embedded devices often do not provide support for floating point computations at all. The floating point operations specified in a high-level programming language are then emulated using arithmetic over machine integers. Emulation of floating point operations is found not only on small embedded systems devices, but also on x86 architectures, where similar methods are used to implement operations on floating point formats for which there is no native support by the hardware (e.g., 128-bit floating point numbers). A key problem is thus to extract floating point abstractions from such program fragments that emulate IEEE-754 operations. We are not aware of the existence of any techniques to tackle this problem.

Bibliography

[1] F. A. Aloul, I. L. Markov, and K. A. Sakallah. Faster SAT and Smaller BDDs via Common Function Structure. In *ICCAD*, pages 443–448, 2001.

[2] T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Two Classes of Boolean Functions for Dependency Analysis. *Sci. Comp. Program.*, 31(1): 3–45, 1998.

[3] R. Bagnara. Personal communication with R. Bagnara via e-mail, May 2010.

[4] R. Bagnara, E. Rodríguez-Carbonell, and E. Zaffanella. Generation of Basic Semi-algebraic Invariants Using Convex Polyhedra. In *SAS*, volume 3672 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2005.

[5] R. Bagnara, K. Dobson, P. M. Hill, M. Mundell, and E. Zaffanella. Grids: A Domain for Analyzing the Distribution of Numerical Values. In *LOPSTR*, volume 4407 of *Lecture Notes in Computer Science*, pages 219–235. Springer, 2006.

[6] R. Bagnara, P. M. Hill, and E. Zaffanella. An Improved Tight Closure Algorithm for Integer Octagonal Constraints. In *VMCAI*, volume 4905 of *Lecture Notes in Computer Science*, pages 8–21. Springer, 2008.

[7] R. Bagnara, P. M. Hill, and E. Zaffanella. Weakly-relational Shapes for Numeric Abstractions: Improved Algorithms and Proofs of Correctness. *Formal Methods in System Design*, 35(3):279–323, 2009.

[8] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008. ISBN 978-0-262-02649-9.

[9] G. Balakrishnan and T. W. Reps. WYSINWYX: What You See Is Not What You eXecute. *ACM Trans. Program. Lang. Syst.*, 32(6), 2010.

[10] G. Balakrishnan, T. Reps, N. Kidd, A. Lal, J. Lim, D. Melski, R. Gruian, S.-H. Yong, C. H. Chen, and T. Teitelbaum. Model Checking x86 Executables with CodeSurfer/x86 and WPDS++. In *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 158–163. Springer, 2005.

[11] G. Balakrishnan, S. Sankaranarayanan, F. Ivancic, O. Wei, and A. Gupta. SLR: Path-Sensitive Analysis through Infeasible-Path Detection and Syntactic Language Refinement. In *SAS*, volume 5079 of *Lecture Notes in Computer Science*, pages 238–254. Springer, 2008.

[12] G. Balakrishnan, S. Sankaranarayanan, F. Ivancic, and A. Gupta. Refining the Control Structure of Loops using Static Analysis. In *EMSOFT*, pages 49–58. ACM Press, 2009.

[13] T. Ball, M. Naik, and S. K. Rajamani. From Symptom to Cause: Localizing Errors in Counterexample Traces. In *POPL*, pages 97–105. ACM Press, 2003.

[14] T. Ball, B. Cook, S. K. Lahiri, and L. Zhang. Zapato: Automatic Theorem Proving for Predicate Abstraction Refinement. In *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 457–461. Springer, 2004.

[15] T. Ball, O. Kupferman, and M. Sagiv. Leaping Loops in the Presence of Abstraction. In *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 491–503. Springer, 2007.

[16] S. Bardin. Personal communication with S. Bardin via e-mail, October 2011.

[17] S. Bardin and P. Herrmann. Structural Testing of Executables. In *ICST*, pages 22–31. IEEE Computer Society Press, 2008.

[18] S. Bardin and P. Herrmann. OSMOSE: Automatic Structural Testing of Executables. *Softw. Test., Verif. & Reliab.*, 21(1):29–54, 2011.

[19] S. Bardin, P. Hermann, J. Leroux, O. Ly, R. Tabary, and A. Vincent. The BINCOA Framework for Binary Code Analysis. In *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 165–170. Springer, 2011.

[20] S. Bardin, P. Herrmann, and F. Védrine. Refinement-Based CFG Reconstruction from Unstructured Programs. In *VMCAI*, volume 6538 of *Lecture Notes in Computer Science*, pages 54–69. Springer, 2011.

[21] E. Barrett and A. King. Range and Set Abstraction using SAT. *Electron. Notes Theor. Comput. Sci.*, 267:17–27, 2010. NSAD.

[22] E. Beckschulze, J. Brauer, A. Stollenwerk, and S. Kowalewski. Analyzing Embedded Systems Code for Mixed-Critical Systems using Hybrid Memory Representations. In *ISORCW/AMICS*, pages 33–40. IEEE Computer Society Press, 2011.

[23] E. A. Bender. *Mathematical Methods in Artificial Intelligence*. IEEE Computer Society Press, 1996.

[24] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. R. Engler. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM*, 53 (2):66–75, 2010.

[25] S. Biallas, J. Brauer, D. Gückel, and S. Kowalewski. On-The-Fly Path Reduction. *Electr. Notes Theor. Comput.Sci.*, 274:3–16, 2010. TTSS.

[26] S. Biallas, J. Brauer, and S. Kowalewski. Counterexample-Guided Abstraction Refinement for PLCs. In *SSV*, pages 1–9. USENIX Association, 2010.

[27] S. Biallas, J. Brauer, S. Kowalewski, and B. Schlich. Automatically Deriving symbolic Invariants for PLC Programs Written in IL. In *FORMS/FORMAT*, pages 237–245. Springer, 2010.

[28] S. Biallas, J. Brauer, and S. Kowalewski. SAT-Based Abstraction Refinement for Programmable Logic Controllers. In *DCDS*, pages 96–101. IEEE Computer Society Press, 2011.

[29] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *TACAS*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[30] A. Blake. *Canonical Expressions in Boolean Algebra*. University of Chicago, 1938.

[31] J. Brauer and A. King. Automatic Abstraction for Intervals using Boolean Formulae. In *SAS*, volume 6337 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 2010.

[32] J. Brauer and A. King. Transfer Function Synthesis without Quantifier Elimination. In *ESOP*, volume 6602 of *Lecture Notes in Computer Science*, pages 97–115. Springer, 2011.

[33] J. Brauer and A. King. Approximate Quantifier Elimination for Propositional Boolean Formulae. In *NFM*, volume 6617 of *Lecture Notes in Computer Science*, pages 73–88. Springer, 2011.

[34] J. Brauer and A. King. Transfer Function Synthesis without Quantifier Elimination. *Logical Methods in Computer Science*, 8(3), 2012.

[35] J. Brauer and A. Simon. Inferring Definite Counterexamples Through Under-Approximation. In *NFM*, volume 7226 of *Lecture Notes in Computer Science*, pages 54–69. Springer, 2012.

[36] J. Brauer, R. Huuck, and B. Schlich. Interprocedural Pointer Analysis in Goanna. *Electr. Notes Theor. Comput. Sci.*, 254:45–63, 2009. SSV.

[37] J. Brauer, B. Schlich, T. Reinbacher, and S. Kowalewski. Stack Bounds Analysis for Microcontroller Assembly Code. In *WESS*. ACM Press, 2009.

[38] J. Brauer, V. Kamin, S. Kowalewski, and T. Noll. Loop Refinement using Octagons and Satisfiability. In *SSV*, pages 1–9. USENIX Association, 2010.

[39] J. Brauer, A. King, and S. Kowalewski. Range Analysis of Microcontroller Code Using Bit-Level Congruences. In *FMICS*, volume 6371 of *Lecture Notes in Computer Science*, pages 82–98. Springer, 2010.

[40] J. Brauer, T. Noll, and B. Schlich. Interval Analysis of Microcontroller Code using Abstract Interpretation of Hardware and Software. In *SCOPES*. ACM Press, 2010.

[41] J. Brauer, R. R. Hansen, S. Kowalewski, K. G. Larsen, and M. Chr. Olesen. Adaptable Value-Set Analysis for Low-Level Code. In *SSV*, 2011. To appear.

[42] J. Brauer, A. King, and J. Kriener. Existential Quantification as Incremental SAT. In *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 191–207. Springer, 2011.

[43] J. Brauer, A. King, and S. Kowalewski. Abstract Interpretation of Microcontroller Code: Intervals Meet Congruences. *Sci. Comp. Program.*, 78(7): 862–883, 2013.

[44] R. Brummayer and A. Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *TACAS*, volume 5505 of *Lecture Notes in Computer Science*, pages 174–177. Springer, 2009.

[45] R. Brummayer, F. Lonsing, and A. Biere. Automated Testing and Debugging of SAT and QBF Solvers. In *SAT*, volume 6175 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2010.

[46] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. The MathSAT 4 SMT Solver. In *CAV*, volume 5123 of *Lecture Notes in Computer Science*, pages 299–303. Springer, 2008.

[47] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[48] R. E. Bryant. Boolean Analysis of MOS Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(4):634–649, 1987.

[49] R. E. Bryant. On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication. *IEEE Trans. Computers*, 40(2):205–213, 1991.

[50] R. E. Bryant. A View from the Engine Room: Computational Support for Symbolic Model Checking. In *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 145–149. Springer, 2008.

[51] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10ˆ20 States and Beyond. In *LICS*, pages 428–439. IEEE Computer Society Press, 1990.

[52] J. R. Burch, E. M. Clarke, and K. L. McMillan. Symbolic Model Checking: 10ˆ20 States and Beyond. *Information and Computation*, 98:142–170, 1992.

[53] S. Bygde. Static WCET Analysis based on Abstract Interpretation and Counting of Elements. Licentiate thesis, Mälardalen University Press, March 2010.

[54] S. Bygde, B. Lisper, and N. Holsti. Fully Bounded Polyhedral Analysis of Integers with Wrapping. In *NSAD*, 2011. To appear.

[55] M. Cadoli, A. Giovanardi, and M. Schaerf. An Algorithm to Evaluate Quantified Boolean Formulae. In *AAAI/IAAI*, pages 262–267. AAAI Press / The MIT Press, 1998.

[56] R. Cavada, A. Cimatti, A. Franzén, K. Kalyanasundaram, M. Roveri, and R. K. Shyamasundar. Computing Predicate Abstractions by Integrating BDDs and SMT Solvers. In *FMCAD*, pages 69–76. IEEE Computer Society Press, 2007.

[57] V. Chandru and J.-L. Lassez. Qualitative Theorem Proving in Linear Constraints. In *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, pages 395–406. Springer, 2003.

[58] B.-Y. E. Chang, M. Harren, and G. C. Necula. Analysis of Low-Level Code Using Cooperating Decompilers. In *SAS*, volume 4134 of *Lecture Notes in Computer Science*, pages 318–335. Springer, 2006.

[59] P. Chauhan, E. M. Clarke, and D. Kroening. A SAT-Based Algorithm for Reparameterization in Symbolic Simulation. In *DAC*, pages 524–529. ACM Press, 2004.

[60] C. Cifuentes and A. Fraboulet. Intraprocedural Static Slicing of Binary Executables. In *ICSM*, pages 188–195. IEEE Computer Society Press, 1997.

[61] C. Cifuentes and M. van Emmerik. Recovery of Jump Table Case Statements from Binary Code. In *IWPC*, pages 192–199. IEEE Computer Society Press, 1999.

[62] R. Clarisó and J. Cortadella. The Octahedron Abstract Domain. *Sci. Comput. Program.*, 64(1):115–139, 2007.

[63] E. M. Clarke and H. Veith. Counterexamples Revisited: Principles, Algorithms, Applications. In *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, pages 208–224. Springer, 2003.

[64] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999. ISBN 0-262-03270-8.

[65] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[66] E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[67] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Progress on the State Explosion Problem in Model Checking. In *Informatics - 10 Years Back. 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 176–194. Springer, 2001.

[68] E. M. Clarke, M. Talupur, H. Veith, and D. Wang. SAT Based Predicate Abstraction for Hardware Verification. In *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 78–92. Springer, 2003.

[69] E. M. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[70] M. Codish. Worst-Case Groundness Analysis Using Positive Boolean Functions. *J. Log. Program.*, 41(1):125–128, 1999.

[71] M. Codish, V. Lagoon, and P. J. Stuckey. Logic Programming with Satisfiability. *TPLP*, 8(1):121–128, 2008.

[72] H. Cohen. BuDDy. Available online at `http://sourceforge.net/projects/buddy`, 2011. Visited: April 2012.

[73] M. Colón. Approximating the Algebraic Relational Semantics of Imperative Programs. In *SAS*, volume 3148 of *Lecture Notes in Computer Science*, pages 296–311. Springer, 2004.

[74] M. Colón and S. Sankaranarayanan. Generalizing the Template Polyhedral Domain. In *ESOP*, volume 6602 of *Lecture Notes in Computer Science*, pages 176–195. Springer, 2011.

[75] B. Cook, D. Kroening, P. Rümmer, and C. Wintersteiger. Ranking Function Synthesis for Bit-Vector Relations. In *TACAS*, volume 6015 of *Lecture Notes in Computer Science*, pages 236–250. Springer, 2010.

[76] O. Coudert and J. C. Madre. Implicit and Incremental Computation of Primes and Essential Primes of Boolean Functions. In *DAC*, pages 36–39. IEEE Computer Society Press, 1992.

[77] 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. ACM Press, 1977.

[78] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *POPL*, pages 269–282. ACM Press, 1979.

[79] P. Cousot and R. Cousot. Comparing the Galois Connection and Widening/-Narrowing Approaches to Abstract Interpretation. In *PLILP*, volume 631 of *Lecture Notes in Computer Science*, pages 269–295. Springer, 1992.

[80] P. Cousot and R. Cousot. Abstract Interpretation Frameworks. *J. Log. Comput.*, 2(4):511–547, 1992.

[81] P. Cousot and R. Cousot. An Abstract Interpretation Framework for Termination. In *POPL*, pages 245–258. ACM Press, 2012.

[82] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–97. ACM, 1978.

[83] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Mine, D. Monniaux, and X. Rival. The Astrée Analyser. In *ESOP*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer, 2005.

[84] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of Abstractions in the ASTRÉE Static Analyzer. In *ASIAN*, volume 4435 of *Lecture Notes in Computer Science*, pages 272–300. Springer, 2006.

[85] P. Cousot, R. Cousot, J. Feret, A. Miné, L. Mauborgne, D. Monniaux, and X. Rival. Varieties of Static Analyzers: A Comparison with ASTREE. In *TASE*, pages 3–20. IEEE Computer Society, 2007.

[86] P. Cousot, R. Cousot, and L. Mauborgne. The Reduced Product of Abstract Domains and the Combination of Decision Procedures. In *FOSSACS*, volume 6604 of *Lecture Notes in Computer Science*, pages 456–472. Springer, 2011.

[87] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.*, pages 451–590, 1991.

[88] R. F. Damiano and J. H. Kukula. Checking Satisfiability of a Conjunction of BDDs. In *DAC*, pages 818–823. ACM, 2003.

[89] S. Das, D. L. Dill, and S. Park. Experience with Predicate Abstraction. In *CAV*, volume 1633 of *Lecture Notes in Computer Science*, pages 160–171. Springer, 1999.

[90] L. de Moura and N. Bjørner. Z3: An Efficient Smt Solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[91] B. De Sutter, B. De Bus, K. De Bosschere, P. Keyngnaert, and B. Demoen. On the Static Analysis of Indirect Control Transfers in Binaries. In *PDPTA*, pages 1013–1019, 2000.

[92] T. Dean and M. Boddy. An Analysis of Time-Dependent Planning. In *AAAI*, pages 49–54. AAAI Press / The MIT Press, 1988.

[93] V. D'Silva, L. Haller, D. Kroening, and M. Tautschnig. Numeric Bounds Analysis with Conflict-Driven Learning. In *TACAS*, volume 7214 of *Lecture Notes in Computer Science*, pages 48–63. Springer, 2012.

[94] N. Eén and N. Sörensson. An Extensible SAT-Solver. In *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[95] N. Eén and N. Sörensson. Translating Pseudo-Boolean Constraints into SAT. *JSAT*, 2(1-4):1–26, 2006.

[96] M. Elder, J. Lim, T. Sharma, T. Andersen, and T. W. Reps. Abstract Domains of Affine Relations. In *SAS*, volume 6887 of *Lecture Notes in Computer Science*, pages 198–215. Springer, 2011.

[97] E. A. Emerson and J. Y. Halpern. "Sometimes" and "Not Never" Revisited: On Branching Versus Linear Time. In *POPL*, pages 127–140, 1983.

[98] G. Filé and F. Ranzato. Improving Abstract Interpretation by Systematic Lifting to the Powerset. In *SLP*, pages 655–669, 1994.

[99] C. Flanagan and S. Qadeer. Predicate Abstraction for Software Verification. In *POPL*, pages 191–202. ACM Press, 2002.

[100] A. Flexeder, B. Mihaila, M. Petter, and H. Seidl. Interprocedural Control Flow Reconstruction. In *APLAS*, volume 6461 of *Lecture Notes in Computer Science*, pages 188–203. Springer, 2010.

[101] C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. SAT Solving for Termination Analysis with Polynomial Interpretations. In *SAT*, volume 4501 of *Lecture Notes in Computer Science*, pages 340–354. Springer, 2007.

[102] D. Geist and I. Beer. Efficient Model Checking by Automated Ordering of Transition Relation Partitions. In *CAV*, volume 818 of *Lecture Notes in Computer Science*, pages 299–310. Springer, 1994.

[103] S. Genaim and F. Spoto. Information Flow Analysis for Java Bytecode. In *VMCAI*, volume 3385 of *Lecture Notes in Computer Science*, pages 346–362, 2005.

[104] S. Genaim, R. Giacobazzi, and I. Mastroeni. Modeling Secure Information Flow with Boolean Functions. In *IFIP WG 1.7, ACM Workshop on Issues in the Theory of Security*, pages 55–66, Barcelona, Spain, 2004.

[105] R. Giacobazzi and E. Quintarelli. Incompleteness, Counterexamples, and Refinements in Abstract Model-Checking. In *SAS*, volume 2126 of *Lecture Notes in Computer Science*, pages 356–373. Springer, 2001.

[106] R. Giacobazzi and F. Ranzato. Optimal Domains for Disjunctive Abstract Interpretation. *Sci. Comput. Program.*, 32(1–3):177–310, 1998.

[107] R. Giacobazzi and F. Scozzari. Intuitionistic Implication in Abstract Interpretation. In *PLILP*, volume 1292 of *Lecture Notes in Computer Science*, pages 175–189. Springer, 1997.

[108] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making Abstract Interpretations Complete. *J. ACM*, 47(2):361–416, 2000.

[109] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making Abstract Domains Condensing. *ACM Trans. Comput. Log.*, 6(1):33–60, 2005.

[110] J. Giesl, P. Schneider-Kamp, and R. Thiemann. Automatic Termination Proofs in the Dependency Pair Framework. In *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 281–286. Springer, 2006.

[111] P. Godefroid and A. Taly. Automated Synthesis of Symbolic Instruction Encodings from I/O Samples. In *PLDI*. ACM Press, 2012.

[112] L. Gonnord and N. Halbwachs. Combining Widening and Acceleration in Linear Relation Analysis. In *SAS*, volume 4134 of *Lecture Notes in Computer Science*, pages 144–160. Springer, 2006.

[113] D. Gopan and T. W. Reps. Lookahead Widening. In *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 452–466. Springer, 2006.

[114] P. Granger. Static Analysis of Arithmetical Congruences. *International Journal of Computer Mathematics*, 30(13):165–190, 1989.

[115] P. Granger. Static Analysis of Linear Congruence Equalities among Variables of a Program. In *TAPSOFT 1991*, volume 493 of *Lecture Notes in Computer Science*, pages 169–192. Springer, 1991.

[116] P. Granger. Static Analyses of Congruence Properties on Rational Numbers. In *SAS*, volume 1302 of *Lecture Notes in Computer Science*, pages 278–292. Springer, 1997.

[117] D. Grund and J. Reineke. Abstract Interpretation of FIFO Replacement. In *SAS*, volume 5673 of *Lecture Notes in Computer Science*, pages 120–136. Springer, 2009.

[118] D. Gückel, J. Brauer, and S. Kowalewski. A System for Synthesizing Abstraction-Enabled Simulators for Binary Code Verification. In *SIES*, pages 118–127. IEEE Computer Society Press, 2010.

[119] D. Gückel, B. Schlich, J. Brauer, and S. Kowalewski. Synthesizing Simulators for Model Checking Microcontroller Binary Code. In *DDECS*, pages 313–316. IEEE Computer Society Press, 2010.

[120] B. S. Gulavani and S. K. Rajamani. Counterexample Driven Refinement for Abstract Interpretation. In *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 474–488. Springer, 2006.

[121] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. SYNERGY: A New Algorithm for Property Checking. In *FSE*, pages 117–127. ACM Press, 2006.

[122] B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. Automatically Refining Abstract Interpretations. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 443–458. Springer, 2008.

[123] S. Gulwani and G. C. Necula. Discovering Affine Aqualities using Random Interpretation. In *POPL*, pages 74–84. ACM Press, 2003.

[124] S. Gulwani and G. C. Necula. Precise Interprocedural Analysis using Random Interpretation. In *POPL*, pages 324–337. ACM Press, 2005.

[125] S. Gulwani, S. Srivastava, and R. Venkatesan. Program Analysis as Constraint Solving. In *PLDI*, pages 281–292. ACM Press, 2008.

[126] A. Gupta, Z. Yang, P. Ashar, and A. Gupta. SAT-Based Image Computation with Application in Reachability Analysis. In *FMCAD*, volume 1954 of *Lecture Notes in Computer Science*, pages 354–371. Springer, 2000.

[127] N. Halbwachs. *Détermination Automatique de Relations Linéaires Vérifiées par les Variables d'un Programme*. PhD thesis, Universit'e Scientifique et Médicale de Grenoble, 1979. `http://www-verimag.imag.fr/~halbwach/bib.html`.

[128] Hex-Rays SA. IdaPro. `http://hex-rays.com/idapro/`. Visited: April 2012.

[129] N. Holsti. Analysing Switch-Case Tables by Partial Evaluation. In *WCET*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[130] J. M. Howe, A. King, and L. Lu. Analysing L ogic Programs by Reasoning Backwards. In *Program Development in Computational Logic*, volume 3049 of *Lecture Notes in Computer Science*, pages 152–188. Springer, 2004.

[131] H. Jin and F. Somenzi. CirCUs: A Hybrid Satisfiability Solver. In *SAT*, volume 3542 of *Lecture Notes in Computer Science*, pages 211–223. Springer, 2005.

[132] N. Jones. Arrays of Pointers to Functions. *Embedded Systems Programming Magazine*, 05:46–56, May 1999.

[133] Y. Jung, J. Kim, J. Shin, and K. Yi. Taming False Alarms from a Domain-Unaware C Analyzer by a Bayesian Statistical Post Analysis. In *SAS*, volume 3672 of *Lecture Notes in Computer Science*, pages 203–217. Springer, 2005.

[134] J. B. Kam and J. D. Ullman. Monotone Data Flow Analysis Frameworks. *Acta Informatica*, 7:305–317, 1976.

[135] D. Kapur. Automatically Generating Loop Invariants Using Quantifier Elimination. In *Deduction and Applications*, volume 05431. IBFI, 2005.

[136] M. Karr. Affine Relationships among Variables of a Program. *Acta Informatica*, 6:133–151, 1976.

[137] N. Kettle. *Anytime Algorithms for ROBDD Symmetry Detection and Approximation*. PhD thesis, Computing Laboratory, 2008.

[138] N. Kettle, A. King, and T. Strzemecki. Widening ROBBDs with Prime Implicants. In *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 105–119. Springer, 2006.

[139] Y. Kim, J. Lee, H. Han, and K.-M. Choe. Filtering False Alarms of Buffer Overflow Analysis using SMT Solvers. *Inform. & Softw. Techn.*, 52(2):210–219, 2010.

[140] J. Kinder and D. Kravchenko. Alternating Control Flow Reconstruction. In *VMCAI*, volume 7148 of *Lecture Notes in Computer Science*, pages 267–282. Springer, 2012.

[141] J. Kinder and H. Veith. Jakstab: A Static Analysis Platform for Binaries. In *CAV*, volume 5123 of *Lecture Notes in Computer Science*, pages 423–427. Springer, 2008.

[142] J. Kinder, H. Veith, and F. Zuleger. An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries. In *VMCAI*, volume 5403 of *Lecture Notes in Computer Science*, pages 214–228. Springer, 2009.

[143] A. King and L. Lu. Forward versus Backward Verification of Logic Programs. In *ICLP*, volume 2916 of *Lecture Notes in Computer Science*, pages 315–330. Springer, 2003.

[144] A. King and H. Søndergaard. Inferring Congruence Equations Using SAT. In *CAV*, volume 5123 of *Lecture Notes in Computer Science*, pages 281–293. Springer, 2008.

[145] A. King and H. Søndergaard. Automatic Abstraction for Congruences. In *VMCAI*, volume 5944 of *Lecture Notes in Computer Science*, pages 197–213. Springer, 2010.

[146] D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 1997.

[147] T. Kremenek and D. R. Engler. Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations. In *SAS*, volume 2694 of *Lecture Notes in Computer Science*, pages 295–315. Springer, 2003.

[148] D. Kroening and O. Strichman. *Decision Procedures*. Springer, 2008.

[149] D. Kroening, A. Groce, and E. M. Clarke. Counterexample Guided Abstraction Refinement Via Program Execution. In *ICFEM*, volume 3308 of *Lecture Notes in Computer Science*, pages 224–238. Springer, 2004.

[150] D. Kroening, N. Sharygina, S. Tonetta, A. Tsitovich, and C. M. Wintersteiger. Loop Summarization Using Abstract Transformers. In *ATVA*, volume 5311 of *Lecture Notes in Computer Science*, pages 111–125. Springer, 2008.

[151] D. Kroening, N. Sharygina, S. Tonetta, A. Tsitovich, and C. M. Wintersteiger. Loopfrog: A Static Analyzer for ANSI-C Programs. In *ASE*, pages 668–670. IEEE Computer Society Press, 2009.

[152] S. K. Lahiri, R. E. Bryant, and B. Cook. A Symbolic Approach to Predicate Abstraction. In *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 141–153. Springer, 2003.

[153] S. K. Lahiri, R. Nieuwenhuis, and A. Oliveras. SMT Techniques for Fast Predicate Abstraction. In *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 424–437. Springer, 2006.

[154] W. Lee, W. Lee, and K. Yi. Sound Non-Statistical Clustering of Static Analysis Alarms. In *VMCAI*, volume 7148 of *Lecture Notes in Computer Science*, pages 299–314. Springer, 2012.

[155] J. Leroux and G. Sutre. Accelerated Data-Flow Analysis. In *SAS*, volume 4634 of *Lecture Notes in Computer Science*, pages 184–199. Springer, 2007.

[156] J. Leroux and G. Sutre. Acceleration in Convex Data-Flow Analysis. In *FSTTCS*, volume 4855 of *Lecture Notes in Computer Science*, pages 520–531. Springer, 2007.

[157] F. Logozzo and M. Fähndrich. Pentagons: A Weakly Relational Abstract Domain for the Efficient Validation of Array Accesses. *Sci. Comput. Program.*, 75(9):796–807, 2010.

[158] L. Lu and A. King. Backward Pair Sharing Analysis. In *FLOPS*, volume 2998 of *Lecture Notes in Computer Science*, pages 132–146. Springer, 2004.

[159] Y. S. Mahajan, Z. Fu, and S. Malik. zChaff: An Efficient SAT Solver. In *SAT*, volume 3542 of *Lecture Notes in Computer Science*, pages 360–375. Springer, 2004.

[160] M. J. Maher. Abduction of Linear Arithmetic Constraints. In *ICLP*, volume 3668 of *Lecture Notes in Computer Science*, pages 174–188. Springer, 2005.

[161] M. J. Maher. Heyting Domains for Constraint Abduction. In *AI*, volume 4304 of *Lecture Notes in Computer Science*, pages 9–18. Springer, 2006.

[162] M. J. Maher and G. Huang. On Computing Constraint Abduction Answers. In *LPAR*, volume 5330 of *Lecture Notes in Computer Science*, pages 421–435. Springer, 2008.

[163] V. M. Manquinho, P. F. Flores, J. P. Marques Silva, and A. L. Oliveira. Prime Implicant Computation Using Satisfiability Algorithms. In *ICTAI*, pages 232–239. IEEE Computer Society Press, 1997.

[164] K. L. McMillan. Applying SAT Methods in Unbounded Symbolic Model Checking. In *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 250–264. Springer, 2002.

[165] P. B. Miltersen, J. Radhakrishnan, and I. Wegener. On Converting CNF to DNF. *Theor. Comput. Sci.*, 347(1-2):325–335, 2005.

[166] A. Miné. The Octagon Abstract Domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[167] D. Monniaux. Automatic Modular Abstractions for Linear Constraints. In *POPL*, pages 140–151. ACM Press, 2009.

[168] D. Monniaux. Quantifier Elimination by Lazy Model Enumeration. In *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 585–599. Springer, 2010.

[169] D. Monniaux. Automatic Modular Abstractions for Template Numerical Constraints. *Logical Methods in Computer Science*, 6(3), 2010.

[170] M. Müller-Olm and H. Seidl. A Note on Karr's Algorithm. In *ICALP*, volume 3142 of *Lecture Notes in Computer Science*, pages 1016–1028. Springer, 2004.

[171] M. Müller-Olm and H. Seidl. Analysis of Modular Arithmetic. In *ESOP*, volume 3444 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 2005.

[172] M. Müller-Olm and H. Seidl. Analysis of Modular Arithmetic. *ACM Trans. Program. Lang. Syst.*, 29(5), August 2007.

[173] M. Müller-Olm and H. Seidl. Upper Adjoints for Fast Inter-procedural Variable Equalities. In *ESOP*, volume 4960 of *Lecture Notes in Computer Science*, pages 178–192. Springer, 2008.

[174] R. Muth, S. K. Debray, S. A. Watterson, and K. De Bosschere. Alto: A Link-Time Optimizer for the Compaq Alpha. *Softw., Pract. Exper.*, 31(1): 67–101, 2001.

[175] H. Oh, K. Heo, W. Lee, W. Lee, and K. Yi. Design and Implementation of Sparse Global Analyses for C-like Languages. In *PLDI*. ACM Press, 2012. to appear.

[176] R. Piskac and T. Wies. Decision Procedures for Automating Termination Proofs. In *VMCAI*, volume 6538 of *Lecture Notes in Computer Science*, pages 371–386. Springer, 2011.

[177] C. Pizzuti. Computing Prime Implicants by Integer Programming. In *ICTAI*, pages 332–336, 1996.

[178] D. A. Plaisted and S. Greenbaum. A Structure-Preserving Clause Form Translation. *Journal of Symbolic Computation*, 2(3):293–304, September 1986.

[179] D. A. Plaisted, A. Biere, and Y. Zhu. A Satisfiability Procedure for Quantified Boolean Formulae. *Discrete Applied Mathematics*, 130(2):291–328, 2003.

[180] PLCopen. Safety Software, Technical Specification, Part 1: Concepts and Function Blocks. online, 2006.

[181] A. Pnueli. The Temporal Logic of Programs. In *FOCS*, pages 46–57. IEEE Computer Society Press, 1977.

[182] W. V. Quine. A Way to Simplify Truth Functions. *American Mathematical Monthly*, 62(9):627–631, 1955.

[183] R. C. Read. Everyone a Winner. *Annals of Discrete Mathematics*, 2:107–120, 1978.

[184] J. Regehr. Who Verifies the Verifiers? `http://blog.regehr.org/archives/370`, 2011. Visited: April 2012.

[185] J. Regehr and U. Duongsaa. Deriving Abstract Transfer Functions for Analyzing Embedded Software. In *LCTES*, pages 34–43. ACM Press, 2006.

[186] J. Regehr and A. Reid. HOIST: A System for Automatically Deriving Static Analyzers for Embedded Systems. *ACM SIGOPS Operating Systems Review*, 38(5):133–143, 2004.

[187] T. Reinbacher and J. Brauer. Precise Control Flow Recovery Using Boolean Logic. In *EMSOFT*, pages 117–126. ACM Press, 2011.

[188] T. Reinbacher, J. Brauer, M. Horauer, and B. Schlich. Refining Assembly Code Static Analysis for the Intel MCS-51 Microcontroller. In *SIES*, pages 161–170. IEEE Computer Society Press, 2009.

[189] T. Reinbacher, M. Horauer, B. Schlich, J. Brauer, and F. Scheuer. Model Checking Assembly Code of an Industrial Knitting Machine. In *EM-COM*, pages 1–8. IEEE Computer Society Press, 2009.

[190] T. Reinbacher, J. Brauer, M. Horauer, A. Steininger, and S. Kowalewski. Test-Case Generation for Embedded Binary Code Using Abstract Interpretation. In *MEMICS*, volume 16 of *OASICS*, pages 101–108. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2010.

[191] T. Reinbacher, M. Horauer, B. Schlich, J. Brauer, and F. Scheuer. Model Checking Assembly Code of an Industrial Knitting Machine. *International Journal of Information Technology, Communications and Convergence (IJITCC)*, 2010.

[192] T. Reinbacher, J. Brauer, M. Horauer, A. Steininger, and S. Kowalewski. Past Time LTL Runtime Verification for Microcontroller Binary Code. In *FMICS*, Lecture Notes in Computer Science, pages 37–51. Springer, 2011.

[193] T. Reinbacher, J. Brauer, D. Schachinger, A. Steininger, and S. Kowalewski. Automated Test-Trace Inspection for Microcontroller Binary Code. In *RV*, Lecture Notes in Computer Science. Springer, 2011. To appear.

[194] T. Reinbacher, A. Steininger, T. Müller, J. Brauer, and S. Kowalewski. Hardware Support for Efficient Testing of Embedded Software. In *MESA*, 2011.

[195] T. W. Reps and G. Balakrishnan. Improved Memory-Access Analysis for x86 Executables. In *CC*, volume 4959 of *Lecture Notes in Computer Science*, pages 16–35. Springer, 2008.

[196] T. W. Reps, S. Horwitz, and M. Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL*, pages 49–61. ACM Press, 1995.

[197] T. W. Reps, M. Sagiv, and G. Yorsh. Symbolic Implementation of the Best Transformer. In *VMCAI*, volume 2937 of *Lecture Notes in Computer Science*, pages 252–266. Springer, 2004.

[198] X. Rival. Understanding the Origin of Alarms in Astrée. In *SAS*, volume 3672 of *Lecture Notes in Computer Science*, pages 303–319. Springer, 2005.

[199] X. Rival. Abstract Dependences for Alarm Diagnosis. In *APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 347–363. Springer, 2005.

[200] X. Rival and L. Mauborgne. The Trace Partitioning Abstract Domain. *ACM Trans. Program. Lang. Syst.*, 29(5), 2007.

[201] E. Rodríguez-Carbonell and D. Kapur. An Abstract Interpretation Approach for Automatic Generation of Polynomial Invariants. In *SAS*, volume 3148 of *Lecture Notes in Computer Science*, pages 280–295. Springer, 2004.

[202] E. Rodríguez-Carbonell and D. Kapur. Automatic Generation of Polynomial Invariants of Bounded Degree using Abstract Interpretation. *Sci. Comput. Program.*, 64(1):54–75, 2007.

[203] R. Rugina and M. C. Rinard. Symbolic Bounds Analysis of Pointers, Array Indices, and Accessed Memory Regions. *ACM Trans. Program. Lang. Syst.*, 27(2):185–235, 2005.

[204] H. Saïdi and N. Shankar. Abstract and Model Check While You Prove. In *CAV*, volume 1633 of *Lecture Notes in Computer Science*, pages 443–454. Springer, 1999.

[205] E. W. Samson and B. E. Mills. Circuit Minimization: Algebra and Algorithms for New Boolean Canonical Expressions. Technical Report TR 54-21, United States Air Force, Cambridge Research Lab, 1954.

[206] S. Sankaranarayanan, H. Sipma, and Z. Manna. Constraint based Linear Relations Analysis. In *SAS*, volume 3148 of *Lecture Notes in Computer Science*, pages 53–68. Springer, 2004.

[207] B. Schlich. *Model Checking of Software for Microcontrollers*. Dissertation, RWTH Aachen University, Aachen, Germany, June 2008. URL `http://aib.informatik.rwth-aachen.de/2008/2008-14.pdf`.

[208] B. Schlich. Model Checking of Software for Microcontrollers. *ACM Trans. Embedded Comput. Syst.*, 9(4), 2010.

[209] B. Schlich, J. Brauer, J. Wernerus, and S. Kowalewski. Direct Model Checking of PLC Programs in IL. In *DCDS*, 2009.

[210] B. Schlich, T. Noll, J. Brauer, and L. Brutschy. Reduction of Interrupt Handler Executions for Model Checking Embedded Software. In *HVC*, volume 6405 of *Lecture Notes in Computer Science*, pages 5–20. Springer, 2009.

[211] B. Schlich, J. Brauer, and S. Kowalewski. Application of Static Analyses for State-Space Reduction to the Microcontroller Binary Code. *Sci. Comput. Program.*, 76(2):100–118, 2011.

[212] P. Schrammel and B. Jeannet. Logico-Numerical Abstract Acceleration and Application to the Verification of Data-Flow Programs. In *SAS*, volume 6887 of *Lecture Notes in Computer Science*, pages 233–248. Springer, 2011.

[213] H. Seidl, A. Flexeder, and M. Petter. Analysing All Polynomial Equations in $\mathbb{Z}_{2^w}$. In *SAS*, volume 5079 of *Lecture Notes in Computer Science*, pages 299–314. Springer, 2008.

[214] A. Sepp, B. Mihaila, and A. Simon. Precise Static Analysis of Binaries by Extracting Relational Information. In *WCRE*, pages 357–366. IEEE Computer Society, 2011.

[215] M. Sharir and A. Pnueli. Two Approaches to Interprocedural Data Flow Analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice Hall, 1981.

[216] M. Sheeran and G. Stålmarck. A Tutorial on Stålmarck's Proof Procedure for Propositional Logic. *Formal Methods in System Design*, 16(1):23–58, 2000.

[217] S. Sheng and M. S. Hsiao. Efficient Preimage Computation Using A Novel Success-Driven ATPG. In *DATE*, pages 10822–10827. IEEE, 2003.

[218] J. P. M. Silva. On Computing Minimum Size Prime Implicants. In *International Workshop on Logic Synthesis*, 1997.

[219] A. Simon. *Value-Range Analysis of C Programs*. Springer, August 2008.

[220] A. Simon. Splitting the Control Flow with Boolean Flags. In *SAS*, volume 5079 of *Lecture Notes in Computer Science*, pages 315–331. Springer, 2008.

[221] A. Simon and A. King. Exploiting Sparsity in Polyhedral Analysis. In *SAS*, volume 3672 of *Lecture Notes in Computer Science*, pages 336–351. Springer, 2005.

[222] A. Simon and A. King. Widening Polyhedra with Landmarks. In *APLAS*, volume 4279 of *Lecture Notes in Computer Science*, pages 166–182. Springer, 2006.

[223] A. Simon and A. King. Taming the Wrapping of Integer Arithmetic. In *SAS*, volume 4634 of *Lecture Notes in Computer Science*, pages 121–136. Springer, 2007.

[224] A. Simon, A. King, and J. M. Howe. Two Variables per Linear Inequality as an Abstract Domain. In *LOPSTR*, volume 2664 of *Lecture Notes in Computer Science*, pages 71–89. Springer, 2002.

[225] A. Simon, A. King, and J. M. Howe. The Two Variable Per Inequality Abstract Domain. *Higher-Order and Symbolic Computation*, 23(1):87–143, 2010.

[226] F. Somenzi. CUDD: CU Decision Diagram Package Release 2.4.2. Available online at `http://vlsi.colorado.edu/~fabio/CUDD`, 2011. Visited: April 2012.

[227] A. Srivastava and D. W. Wall. A Practical System for Intermodule Code Optimization at Link-Time. *Journal of Programming Languages*, 1(1):1–18, December 1992.

[228] A. Thakur and T. W. Reps. A Method for Symbolic Computation of Abstract Operations. Technical Report TR-1708, Computer Sciences Department, University of Wisconsin, Madison, WI, January 2012.

[229] A. Thakur and T. W. Reps. A Method for Symbolic Computation of Abstract Operations. In *CAV*, Lecture Notes in Computer Science. Springer, 2012. To appear.

[230] A. Thakur, M. Elder, and T. W. Reps. Bilateral Algorithms for Symbolic Abstraction. Technical Report TR-1713, Computer Sciences Department, University of Wisconsin, Madison, WI, March 2012.

[231] A. V. Thakur, J. Lim, A. Lal, A. Burton, E. Driscoll, M. Elder, T. Andersen, and T. W. Reps. Directed Proof Generation for Machine Code. In *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 288–305. Springer, 2010.

[232] H. Theiling. Extracting Safe and Precise Control Flow from Binaries. In *RTCAS*, pages 23–30. IEEE Computer Society Press, 2000.

[233] R. Thiemann and J. Giesl. Size-Change Termination for Term Rewriting. In *RTA*, volume 2706 of *Lecture Notes in Computer Science*, pages 264–278. Springer, 2003.

[234] G. S. Tseitin. On the Complexity of Derivation in the Propositional Calculus. In A. O. Slisenko, editor, *Studies in Constructive Mathematics and Mathematical Logic*, volume Part II, pages 115–125, 1968.

[235] A. Tsitovich, N. Sharygina, C. M. Wintersteiger, and D. Kroening. Loop Summarization and Termination Analysis. In *TACAS*, volume 6605 of *Lecture Notes in Computer Science*, pages 81–95. Springer, 2011.

[236] C. Umans. The Minimum Equivalent DNF Problem and Shortest Implicants. In *FOCS*, pages 556–563. IEEE Computer Society Press, 1998.

[237] M. Van Emmerik and T. Waddington. Using a Decompiler for Real-World Source Recovery. In *WCRE*, pages 27–36. IEEE Computer Society, 2004.

[238] B. Wachter and L. Zhang. Best Probabilistic Transformers. In *VMCAI*, volume 5944 of *Lecture Notes in Computer Science*, pages 362–379. Springer, 2010.

[239] H. S. Warren. *Hacker's Delight*. Addison-Wesley, 2002.

[240] S. Weaver, J. V. Franco, and J. S. Schlipf. Extending Existential Quantification in Conjunctions of BDDs. *JSAT*, 1(2):89–110, 2006.

[241] V. Weispfenning. Comprehensive Gröbner Bases. *Journal of Symbolic Computation*, 14(1):1–30, 1992.

[242] J. Whittemore, J. Kim, and K. Sakallah. SATIRE: A New Incremental Satisfiability Engine. In *DAC*, pages 542–545. ACM Press, 2001.

[243] C. M. Wintersteiger, Y. Hamadi, and L. de Moura. Efficiently Solving Quantified Bit-Vector Formulas. In *FMCAD*, pages 239–246. IEEE Computer Society Press, 2010.

Aachener Informatik-Berichte

This is the list of all technical reports since 1987. To obtain copies of reports please consult

`http://aib.informatik.rwth-aachen.de/`

or send your request to:

Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen,
Email: biblio@informatik.rwth-aachen.de

1987-01 * Fachgruppe Informatik: Jahresbericht 1986

1987-02 * David de Frutos Escrig, Klaus Indermark: Equivalence Relations of Non-Deterministic Ianov-Schemes

1987-03 * Manfred Nagl: A Software Development Environment based on Graph Technology

1987-04 * Claus Lewerentz, Manfred Nagl, Bernhard Westfechtel: On Integration Mechanisms within a Graph-Based Software Development Environment

1987-05 * Reinhard Rinn: Über Eingabeanomalien bei verschiedenen Inferenzmodellen

1987-06 * Werner Damm, Gert Döhmen: Specifying Distributed Computer Architectures in AADL*

1987-07 * Gregor Engels, Claus Lewerentz, Wilhelm Schäfer: Graph Grammar Engineering: A Software Specification Method

1987-08 * Manfred Nagl: Set Theoretic Approaches to Graph Grammars

1987-09 * Claus Lewerentz, Andreas Schürr: Experiences with a Database System for Software Documents

1987-10 * Herbert Klaeren, Klaus Indermark: A New Implementation Technique for Recursive Function Definitions

1987-11 * Rita Loogen: Design of a Parallel Programmable Graph Reduction Machine with Distributed Memory

1987-12 J. Börstler, U. Möncke, R. Wilhelm: Table compression for tree automata

1988-01 * Gabriele Esser, Johannes Rückert, Frank Wagner Gesellschaftliche Aspekte der Informatik

1988-02 * Peter Martini, Otto Spaniol: Token-Passing in High-Speed Backbone Networks for Campus-Wide Environments

1988-03 * Thomas Welzel: Simulation of a Multiple Token Ring Backbone

1988-04 * Peter Martini: Performance Comparison for HSLAN Media Access Protocols

1988-05 * Peter Martini: Performance Analysis of Multiple Token Rings
1988-06 * Andreas Mann, Johannes Rückert, Otto Spaniol: Datenfunknetze
1988-07 * Andreas Mann, Johannes Rückert: Packet Radio Networks for Data Exchange
1988-08 * Andreas Mann, Johannes Rückert: Concurrent Slot Assignment Protocol for Packet Radio Networks
1988-09 * W. Kremer, F. Reichert, J. Rückert, A. Mann: Entwurf einer Netzwerktopologie für ein Mobilfunknetz zur Unterstützung des öffentlichen Straßenverkehrs
1988-10 * Kai Jakobs: Towards User-Friendly Networking
1988-11 * Kai Jakobs: The Directory - Evolution of a Standard
1988-12 * Kai Jakobs: Directory Services in Distributed Systems - A Survey
1988-13 * Martine Schümmer: RS-511, a Protocol for the Plant Floor
1988-14 * U. Quernheim: Satellite Communication Protocols - A Performance Comparison Considering On-Board Processing
1988-15 * Peter Martini, Otto Spaniol, Thomas Welzel: File Transfer in High Speed Token Ring Networks: Performance Evaluation by Approximate Analysis and Simulation
1988-16 * Fachgruppe Informatik: Jahresbericht 1987
1988-17 * Wolfgang Thomas: Automata on Infinite Objects
1988-18 * Michael Sonnenschein: On Petri Nets and Data Flow Graphs
1988-19 * Heiko Vogler: Functional Distribution of the Contextual Analysis in Block-Structured Programming Languages: A Case Study of Tree Transducers
1988-20 * Thomas Welzel: Einsatz des Simulationswerkzeuges QNAP2 zur Leistungsbewertung von Kommunikationsprotokollen
1988-21 * Th. Janning, C. Lewerentz: Integrated Project Team Management in a Software Development Environment
1988-22 * Joost Engelfriet, Heiko Vogler: Modular Tree Transducers
1988-23 * Wolfgang Thomas: Automata and Quantifier Hierarchies
1988-24 * Uschi Heuter: Generalized Definite Tree Languages
1989-01 * Fachgruppe Informatik: Jahresbericht 1988
1989-02 * G. Esser, J. Rückert, F. Wagner (Hrsg.): Gesellschaftliche Aspekte der Informatik
1989-03 * Heiko Vogler: Bottom-Up Computation of Primitive Recursive Tree Functions
1989-04 * Andy Schürr: Introduction to PROGRESS, an Attribute Graph Grammar Based Specification Language
1989-05 J. Börstler: Reuse and Software Development - Problems, Solutions, and Bibliography (in German)
1989-06 * Kai Jakobs: OSI - An Appropriate Basis for Group Communication?

1989-07 * Kai Jakobs: ISO's Directory Proposal - Evolution, Current Status and Future Problems

1989-08 * Bernhard Westfechtel: Extension of a Graph Storage for Software Documents with Primitives for Undo/Redo and Revision Control

1989-09 * Peter Martini: High Speed Local Area Networks - A Tutorial

1989-10 * P. Davids, Th. Welzel: Performance Analysis of DQDB Based on Simulation

1989-11 * Manfred Nagl (Ed.): Abstracts of Talks presented at the WG '89 15th International Workshop on Graphtheoretic Concepts in Computer Science

1989-12 * Peter Martini: The DQDB Protocol - Is it Playing the Game?

1989-13 * Martine Schümmer: CNC/DNC Communication with MAP

1989-14 * Martine Schümmer: Local Area Networks for Manufactoring Environments with hard Real-Time Requirements

1989-15 * M. Schümmer, Th. Welzel, P. Martini: Integration of Field Bus and MAP Networks - Hierarchical Communication Systems in Production Environments

1989-16 * G. Vossen, K.-U. Witt: SUXESS: Towards a Sound Unification of Extensions of the Relational Data Model

1989-17 * J. Derissen, P. Hruschka, M.v.d. Beeck, Th. Janning, M. Nagl: Integrating Structured Analysis and Information Modelling

1989-18 A. Maassen: Programming with Higher Order Functions

1989-19 * Mario Rodriguez-Artalejo, Heiko Vogler: A Narrowing Machine for Syntax Directed BABEL

1989-20 H. Kuchen, R. Loogen, J.J. Moreno Navarro, M. Rodriguez Artalejo: Graph-based Implementation of a Functional Logic Language

1990-01 * Fachgruppe Informatik: Jahresbericht 1989

1990-02 * Vera Jansen, Andreas Potthoff, Wolfgang Thomas, Udo Wermuth: A Short Guide to the AMORE System (Computing Automata, MOnoids and Regular Expressions)

1990-03 * Jerzy Skurczynski: On Three Hierarchies of Weak SkS Formulas

1990-04 R. Loogen: Stack-based Implementation of Narrowing

1990-05 H. Kuchen, A. Wagener: Comparison of Dynamic Load Balancing Strategies

1990-06 * Kai Jakobs, Frank Reichert: Directory Services for Mobile Communication

1990-07 * Kai Jakobs: What's Beyond the Interface - OSI Networks to Support Cooperative Work

1990-08 * Kai Jakobs: Directory Names and Schema - An Evaluation

1990-09 * Ulrich Quernheim, Dieter Kreuer: Das CCITT - Signalisierungssystem Nr. 7 auf Satellitenstrecken; Simulation der Zeichengabestrecke

1990-11 H. Kuchen, R. Loogen, J.J. Moreno Navarro, M. Rodriguez Artalejo: Lazy Narrowing in a Graph Machine

1990-12 * Kai Jakobs, Josef Kaltwasser, Frank Reichert, Otto Spaniol: Der Computer fährt mit

1990-13 * Rudolf Mathar, Andreas Mann: Analyzing a Distributed Slot Assignment Protocol by Markov Chains

1990-14 A. Maassen: Compilerentwicklung in Miranda - ein Praktikum in funktionaler Programmierung (written in german)

1990-15 * Manfred Nagl, Andreas Schürr: A Specification Environment for Graph Grammars

1990-16 A. Schürr: PROGRESS: A VHL-Language Based on Graph Grammars

1990-17 * Marita Möller: Ein Ebenenmodell wissensbasierter Konsultationen - Unterstützung für Wissensakquisition und Erklärungsfähigkeit

1990-18 * Eric Kowalewski: Entwurf und Interpretation einer Sprache zur Beschreibung von Konsultationsphasen in Expertensystemen

1990-20 Y. Ortega Mallen, D. de Frutos Escrig: A Complete Proof System for Timed Observations

1990-21 * Manfred Nagl: Modelling of Software Architectures: Importance, Notions, Experiences

1990-22 H. Fassbender, H. Vogler: A Call-by-need Implementation of Syntax Directed Functional Programming

1991-01 Guenther Geiler (ed.), Fachgruppe Informatik: Jahresbericht 1990

1991-03 B. Steffen, A. Ingolfsdottir: Characteristic Formulae for Processes with Divergence

1991-04 M. Portz: A new class of cryptosystems based on interconnection networks

1991-05 H. Kuchen, G. Geiler: Distributed Applicative Arrays

1991-06 * Ludwig Staiger: Kolmogorov Complexity and Hausdorff Dimension

1991-07 * Ludwig Staiger: Syntactic Congruences for w-languages

1991-09 * Eila Kuikka: A Proposal for a Syntax-Directed Text Processing System

1991-10 K. Gladitz, H. Fassbender, H. Vogler: Compiler-based Implementation of Syntax-Directed Functional Programming

1991-11 R. Loogen, St. Winkler: Dynamic Detection of Determinism in Functional Logic Languages

1991-12 * K. Indermark, M. Rodriguez Artalejo (Eds.): Granada Workshop on the Integration of Functional and Logic Programming

1991-13 * Rolf Hager, Wolfgang Kremer: The Adaptive Priority Scheduler: A More Fair Priority Service Discipline

1991-14 * Andreas Fasbender, Wolfgang Kremer: A New Approximation Algorithm for Tandem Networks with Priority Nodes

1991-15 J. Börstler, A. Zündorf: Revisiting extensions to Modula-2 to support reusability

1991-16 J. Börstler, Th. Janning: Bridging the gap between Requirements Analysis and Design

1991-17 A. Zündorf, A. Schürr: Nondeterministic Control Structures for Graph Rewriting Systems

1991-18 * Matthias Jarke, John Mylopoulos, Joachim W. Schmidt, Yannis Vassiliou: DAIDA: An Environment for Evolving Information Systems

1991-19 M. Jeusfeld, M. Jarke: From Relational to Object-Oriented Integrity Simplification

1991-20 G. Hogen, A. Kindler, R. Loogen: Automatic Parallelization of Lazy Functional Programs

1991-21 * Prof. Dr. rer. nat. Otto Spaniol: ODP (Open Distributed Processing): Yet another Viewpoint

1991-22 H. Kuchen, F. Lücking, H. Stoltze: The Topology Description Language TDL

1991-23 S. Graf, B. Steffen: Compositional Minimization of Finite State Systems

1991-24 R. Cleaveland, J. Parrow, B. Steffen: The Concurrency Workbench: A Semantics Based Tool for the Verification of Concurrent Systems

1991-25 * Rudolf Mathar, Jürgen Mattfeldt: Optimal Transmission Ranges for Mobile Communication in Linear Multihop Packet Radio Networks

1991-26 M. Jeusfeld, M. Staudt: Query Optimization in Deductive Object Bases

1991-27 J. Knoop, B. Steffen: The Interprocedural Coincidence Theorem

1991-28 J. Knoop, B. Steffen: Unifying Strength Reduction and Semantic Code Motion

1991-30 T. Margaria: First-Order theories for the verification of complex FSMs

1991-31 B. Steffen: Generating Data Flow Analysis Algorithms from Modal Specifications

1992-01 Stefan Eherer (ed.), Fachgruppe Informatik: Jahresbericht 1991

1992-02 * Bernhard Westfechtel: Basismechanismen zur Datenverwaltung in strukturbezogenen Hypertextsystemen

1992-04 S. A. Smolka, B. Steffen: Priority as Extremal Probability

1992-05 * Matthias Jarke, Carlos Maltzahn, Thomas Rose: Sharing Processes: Team Coordination in Design Repositories

1992-06 O. Burkart, B. Steffen: Model Checking for Context-Free Processes

1992-07 * Matthias Jarke, Klaus Pohl: Information Systems Quality and Quality Information Systems

1992-08 * Rudolf Mathar, Jürgen Mattfeldt: Analyzing Routing Strategy NFP in Multihop Packet Radio Networks on a Line

1992-09 * Alfons Kemper, Guido Moerkotte: Grundlagen objektorientierter Datenbanksysteme

1992-10	Matthias Jarke, Manfred Jeusfeld, Andreas Miethsam, Michael Gocek: Towards a logic-based reconstruction of software configuration management
1992-11	Werner Hans: A Complete Indexing Scheme for WAM-based Abstract Machines
1992-12	W. Hans, R. Loogen, St. Winkler: On the Interaction of Lazy Evaluation and Backtracking
1992-13 *	Matthias Jarke, Thomas Rose: Specification Management with CAD
1992-14	Th. Noll, H. Vogler: Top-down Parsing with Simultaneous Evaluation on Noncircular Attribute Grammars
1992-15	A. Schuerr, B. Westfechtel: Graphgrammatiken und Graphersetzungssysteme(written in german)
1992-16 *	Graduiertenkolleg Informatik und Technik (Hrsg.): Forschungsprojekte des Graduiertenkollegs Informatik und Technik
1992-17	M. Jarke (ed.): ConceptBase V3.1 User Manual
1992-18 *	Clarence A. Ellis, Matthias Jarke (Eds.): Distributed Cooperation in Integrated Information Systems - Proceedings of the Third International Workshop on Intelligent and Cooperative Information Systems
1992-19-00	H. Kuchen, R. Loogen (eds.): Proceedings of the 4th Int. Workshop on the Parallel Implementation of Functional Languages
1992-19-01	G. Hogen, R. Loogen: PASTEL - A Parallel Stack-Based Implementation of Eager Functional Programs with Lazy Data Structures (Extended Abstract)
1992-19-02	H. Kuchen, K. Gladitz: Implementing Bags on a Shared Memory MIMD-Machine
1992-19-03	C. Rathsack, S.B. Scholz: LISA - A Lazy Interpreter for a Full-Fledged Lambda-Calculus
1992-19-04	T.A. Bratvold: Determining Useful Parallelism in Higher Order Functions
1992-19-05	S. Kahrs: Polymorphic Type Checking by Interpretation of Code
1992-19-06	M. Chakravarty, M. Köhler: Equational Constraints, Residuation, and the Parallel JUMP-Machine
1992-19-07	J. Seward: Polymorphic Strictness Analysis using Frontiers (Draft Version)
1992-19-08	D. Gärtner, A. Kimms, W. Kluge: pi-Redˆ+ - A Compiling Graph-Reduction System for a Full Fledged Lambda-Calculus
1992-19-09	D. Howe, G. Burn: Experiments with strict STG code
1992-19-10	J. Glauert: Parallel Implementation of Functional Languages Using Small Processes
1992-19-11	M. Joy, T. Axford: A Parallel Graph Reduction Machine
1992-19-12	A. Bennett, P. Kelly: Simulation of Multicache Parallel Reduction

1992-19-13 K. Langendoen, D.J. Agterkamp: Cache Behaviour of Lazy Functional Programs (Working Paper)
1992-19-14 K. Hammond, S. Peyton Jones: Profiling scheduling strategies on the GRIP parallel reducer
1992-19-15 S. Mintchev: Using Strictness Information in the STG-machine
1992-19-16 D. Rushall: An Attribute Grammar Evaluator in Haskell
1992-19-17 J. Wild, H. Glaser, P. Hartel: Statistics on storage management in a lazy functional language implementation
1992-19-18 W.S. Martins: Parallel Implementations of Functional Languages
1992-19-19 D. Lester: Distributed Garbage Collection of Cyclic Structures (Draft version)
1992-19-20 J.C. Glas, R.F.H. Hofman, W.G. Vree: Parallelization of Branch-and-Bound Algorithms in a Functional Programming Environment
1992-19-21 S. Hwang, D. Rushall: The nu-STG machine: a parallelized Spineless Tagless Graph Reduction Machine in a distributed memory architecture (Draft version)
1992-19-22 G. Burn, D. Le Metayer: Cps-Translation and the Correctness of Optimising Compilers
1992-19-23 S.L. Peyton Jones, P. Wadler: Imperative functional programming (Brief summary)
1992-19-24 W. Damm, F. Liu, Th. Peikenkamp: Evaluation and Parallelization of Functions in Functional + Logic Languages (abstract)
1992-19-25 M. Kesseler: Communication Issues Regarding Parallel Functional Graph Rewriting
1992-19-26 Th. Peikenkamp: Charakterizing and representing neededness in functional loginc languages (abstract)
1992-19-27 H. Doerr: Monitoring with Graph-Grammars as formal operational Models
1992-19-28 J. van Groningen: Some implementation aspects of Concurrent Clean on distributed memory architectures
1992-19-29 G. Ostheimer: Load Bounding for Implicit Parallelism (abstract)
1992-20 H. Kuchen, F.J. Lopez Fraguas, J.J. Moreno Navarro, M. Rodriguez Artalejo: Implementing Disequality in a Lazy Functional Logic Language
1992-21 H. Kuchen, F.J. Lopez Fraguas: Result Directed Computing in a Functional Logic Language
1992-22 H. Kuchen, J.J. Moreno Navarro, M.V. Hermenegildo: Independent AND-Parallel Narrowing
1992-23 T. Margaria, B. Steffen: Distinguishing Formulas for Free
1992-24 K. Pohl: The Three Dimensions of Requirements Engineering
1992-25 * R. Stainov: A Dynamic Configuration Facility for Multimedia Communications

1992-26 * Michael von der Beeck: Integration of Structured Analysis and Timed Statecharts for Real-Time and Concurrency Specification

1992-27 W. Hans, St. Winkler: Aliasing and Groundness Analysis of Logic Programs through Abstract Interpretation and its Safety

1992-28 * Gerhard Steinke, Matthias Jarke: Support for Security Modeling in Information Systems Design

1992-29 B. Schinzel: Warum Frauenforschung in Naturwissenschaft und Technik

1992-30 A. Kemper, G. Moerkotte, K. Peithner: Object-Orientation Axiomatised by Dynamic Logic

1992-32 * Bernd Heinrichs, Kai Jakobs: Timer Handling in High-Performance Transport Systems

1992-33 * B. Heinrichs, K. Jakobs, K. Lenßen, W. Reinhardt, A. Spinner: EuroBridge: Communication Services for Multimedia Applications

1992-34 C. Gerlhof, A. Kemper, Ch. Kilger, G. Moerkotte: Partition-Based Clustering in Object Bases: From Theory to Practice

1992-35 J. Börstler: Feature-Oriented Classification and Reuse in IPSEN

1992-36 M. Jarke, J. Bubenko, C. Rolland, A. Sutcliffe, Y. Vassiliou: Theories Underlying Requirements Engineering: An Overview of NATURE at Genesis

1992-37 * K. Pohl, M. Jarke: Quality Information Systems: Repository Support for Evolving Process Models

1992-38 A. Zuendorf: Implementation of the imperative / rule based language PROGRES

1992-39 P. Koch: Intelligentes Backtracking bei der Auswertung funktional-logischer Programme

1992-40 * Rudolf Mathar, Jürgen Mattfeldt: Channel Assignment in Cellular Radio Networks

1992-41 * Gerhard Friedrich, Wolfgang Neidl: Constructive Utility in Model-Based Diagnosis Repair Systems

1992-42 * P. S. Chen, R. Hennicker, M. Jarke: On the Retrieval of Reusable Software Components

1992-43 W. Hans, St.Winkler: Abstract Interpretation of Functional Logic Languages

1992-44 N. Kiesel, A. Schuerr, B. Westfechtel: Design and Evaluation of GRAS, a Graph-Oriented Database System for Engineering Applications

1993-01 * Fachgruppe Informatik: Jahresbericht 1992

1993-02 * Patrick Shicheng Chen: On Inference Rules of Logic-Based Information Retrieval Systems

1993-03 G. Hogen, R. Loogen: A New Stack Technique for the Management of Runtime Structures in Distributed Environments

1993-05 A. Zündorf: A Heuristic for the Subgraph Isomorphism Problem in Executing PROGRES

1993-06 A. Kemper, D. Kossmann: Adaptable Pointer Swizzling Strategies in Object Bases: Design, Realization, and Quantitative Analysis

1993-07 * Graduiertenkolleg Informatik und Technik (Hrsg.): Graduiertenkolleg Informatik und Technik

1993-08 * Matthias Berger: k-Coloring Vertices using a Neural Network with Convergence to Valid Solutions

1993-09 M. Buchheit, M. Jeusfeld, W. Nutt, M. Staudt: Subsumption between Queries to Object-Oriented Databases

1993-10 O. Burkart, B. Steffen: Pushdown Processes: Parallel Composition and Model Checking

1993-11 * R. Große-Wienker, O. Hermanns, D. Menzenbach, A. Pollacks, S. Repetzki, J. Schwartz, K. Sonnenschein, B. Westfechtel: Das SUKITS-Projekt: A-posteriori-Integration heterogener CIM-Anwendungssysteme

1993-12 * Rudolf Mathar, Jürgen Mattfeldt: On the Distribution of Cumulated Interference Power in Rayleigh Fading Channels

1993-13 O. Maler, L. Staiger: On Syntactic Congruences for omega-languages

1993-14 M. Jarke, St. Eherer, R. Gallersdoerfer, M. Jeusfeld, M. Staudt: ConceptBase - A Deductive Object Base Manager

1993-15 M. Staudt, H.W. Nissen, M.A. Jeusfeld: Query by Class, Rule and Concept

1993-16 * M. Jarke, K. Pohl, St. Jacobs et al.: Requirements Engineering: An Integrated View of Representation Process and Domain

1993-17 * M. Jarke, K. Pohl: Establishing Vision in Context: Towards a Model of Requirements Processes

1993-18 W. Hans, H. Kuchen, St. Winkler: Full Indexing for Lazy Narrowing

1993-19 W. Hans, J.J. Ruz, F. Saenz, St. Winkler: A VHDL Specification of a Shared Memory Parallel Machine for Babel

1993-20 * K. Finke, M. Jarke, P. Szczurko, R. Soltysiak: Quality Management for Expert Systems in Process Control

1993-21 M. Jarke, M.A. Jeusfeld, P. Szczurko: Three Aspects of Intelligent Cooperation in the Quality Cycle

1994-01 Margit Generet, Sven Martin (eds.), Fachgruppe Informatik: Jahresbericht 1993

1994-02 M. Lefering: Development of Incremental Integration Tools Using Formal Specifications

1994-03 * P. Constantopoulos, M. Jarke, J. Mylopoulos, Y. Vassiliou: The Software Information Base: A Server for Reuse

1994-04 * Rolf Hager, Rudolf Mathar, Jürgen Mattfeldt: Intelligent Cruise Control and Reliable Communication of Mobile Stations

1994-05 * Rolf Hager, Peter Hermesmann, Michael Portz: Feasibility of Authentication Procedures within Advanced Transport Telematics

1994-06 * Claudia Popien, Bernd Meyer, Axel Kuepper: A Formal Approach to Service Import in ODP Trader Federations

1994-07 P. Peters, P. Szczurko: Integrating Models of Quality Management Methods by an Object-Oriented Repository

1994-08 * Manfred Nagl, Bernhard Westfechtel: A Universal Component for the Administration in Distributed and Integrated Development Environments

1994-09 * Patrick Horster, Holger Petersen: Signatur- und Authentifikationsverfahren auf der Basis des diskreten Logarithmusproblems

1994-11 A. Schürr: PROGRES, A Visual Language and Environment for PROgramming with Graph REwrite Systems

1994-12 A. Schürr: Specification of Graph Translators with Triple Graph Grammars

1994-13 A. Schürr: Logic Based Programmed Structure Rewriting Systems

1994-14 L. Staiger: Codes, Simplifying Words, and Open Set Condition

1994-15 * Bernhard Westfechtel: A Graph-Based System for Managing Configurations of Engineering Design Documents

1994-16 P. Klein: Designing Software with Modula-3

1994-17 I. Litovsky, L. Staiger: Finite acceptance of infinite words

1994-18 G. Hogen, R. Loogen: Parallel Functional Implementations: Graphbased vs. Stackbased Reduction

1994-19 M. Jeusfeld, U. Johnen: An Executable Meta Model for Re-Engineering of Database Schemas

1994-20 * R. Gallersdörfer, M. Jarke, K. Klabunde: Intelligent Networks as a Data Intensive Application (INDIA)

1994-21 M. Mohnen: Proving the Correctness of the Static Link Technique Using Evolving Algebras

1994-22 H. Fernau, L. Staiger: Valuations and Unambiguity of Languages, with Applications to Fractal Geometry

1994-24 * M. Jarke, K. Pohl, R. Dömges, St. Jacobs, H. W. Nissen: Requirements Information Management: The NATURE Approach

1994-25 * M. Jarke, K. Pohl, C. Rolland, J.-R. Schmitt: Experience-Based Method Evaluation and Improvement: A Process Modeling Approach

1994-26 * St. Jacobs, St. Kethers: Improving Communication and Decision Making within Quality Function Deployment

1994-27 * M. Jarke, H. W. Nissen, K. Pohl: Tool Integration in Evolving Information Systems Environments

1994-28 O. Burkart, D. Caucal, B. Steffen: An Elementary Bisimulation Decision Procedure for Arbitrary Context-Free Processes

1995-01 * Fachgruppe Informatik: Jahresbericht 1994

1995-02 Andy Schürr, Andreas J. Winter, Albert Zündorf: Graph Grammar Engineering with PROGRES

1995-03 Ludwig Staiger: A Tight Upper Bound on Kolmogorov Complexity by Hausdorff Dimension and Uniformly Optimal Prediction

1995-04 Birgitta König-Ries, Sven Helmer, Guido Moerkotte: An experimental study on the complexity of left-deep join ordering problems for cyclic queries

1995-05 Sophie Cluet, Guido Moerkotte: Efficient Evaluation of Aggregates on Bulk Types

1995-06 Sophie Cluet, Guido Moerkotte: Nested Queries in Object Bases

1995-07 Sophie Cluet, Guido Moerkotte: Query Optimization Techniques Exploiting Class Hierarchies

1995-08 Markus Mohnen: Efficient Compile-Time Garbage Collection for Arbitrary Data Structures

1995-09 Markus Mohnen: Functional Specification of Imperative Programs: An Alternative Point of View of Functional Languages

1995-10 Rainer Gallersdörfer, Matthias Nicola: Improving Performance in Replicated Databases through Relaxed Coherency

1995-11 * M.Staudt, K.von Thadden: Subsumption Checking in Knowledge Bases

1995-12 * G.V.Zemanek, H.W.Nissen, H.Hubert, M.Jarke: Requirements Analysis from Multiple Perspectives: Experiences with Conceptual Modeling Technology

1995-13 * M.Staudt, M.Jarke: Incremental Maintenance of Externally Materialized Views

1995-14 * P.Peters, P.Szczurko, M.Jeusfeld: Oriented Information Management: Conceptual Models at Work

1995-15 * Matthias Jarke, Sudha Ram (Hrsg.): WITS 95 Proceedings of the 5th Annual Workshop on Information Technologies and Systems

1995-16 * W.Hans, St.Winkler, F.Saenz: Distributed Execution in Functional Logic Programming

1996-01 * Jahresbericht 1995

1996-02 Michael Hanus, Christian Prehofer: Higher-Order Narrowing with Definitional Trees

1996-03 * W.Scheufele, G.Moerkotte: Optimal Ordering of Selections and Joins in Acyclic Queries with Expensive Predicates

1996-04 Klaus Pohl: PRO-ART: Enabling Requirements Pre-Traceability

1996-05 Klaus Pohl: Requirements Engineering: An Overview

1996-06 * M.Jarke, W.Marquardt: Design and Evaluation of Computer–Aided Process Modelling Tools

1996-07	Olaf Chitil: The Sigma-Semantics: A Comprehensive Semantics for Functional Programs
1996-08 *	S.Sripada: On Entropy and the Limitations of the Second Law of Thermodynamics
1996-09	Michael Hanus (Ed.): Proceedings of the Poster Session of ALP96 - Fifth International Conference on Algebraic and Logic Programming
1996-09-0	Michael Hanus (Ed.): Proceedings of the Poster Session of ALP 96 - Fifth International Conference on Algebraic and Logic Programming: Introduction and table of contents
1996-09-1	Ilies Alouini: An Implementation of Conditional Concurrent Rewriting on Distributed Memory Machines
1996-09-2	Olivier Danvy, Karoline Malmkjær: On the Idempotence of the CPS Transformation
1996-09-3	Víctor M. Gulías, José L. Freire: Concurrent Programming in Haskell
1996-09-4	Sébastien Limet, Pierre Réty: On Decidability of Unifiability Modulo Rewrite Systems
1996-09-5	Alexandre Tessier: Declarative Debugging in Constraint Logic Programming
1996-10	Reidar Conradi, Bernhard Westfechtel: Version Models for Software Configuration Management
1996-11 *	C.Weise, D.Lenzkes: A Fast Decision Algorithm for Timed Refinement
1996-12 *	R.Dömges, K.Pohl, M.Jarke, B.Lohmann, W.Marquardt: PRO-ART/CE* — An Environment for Managing the Evolution of Chemical Process Simulation Models
1996-13 *	K.Pohl, R.Klamma, K.Weidenhaupt, R.Dömges, P.Haumer, M.Jarke: A Framework for Process-Integrated Tools
1996-14 *	R.Gallersdörfer, K.Klabunde, A.Stolz, M.Eßmajor: INDIA — Intelligent Networks as a Data Intensive Application, Final Project Report, June 1996
1996-15 *	H.Schimpe, M.Staudt: VAREX: An Environment for Validating and Refining Rule Bases
1996-16 *	M.Jarke, M.Gebhardt, S.Jacobs, H.Nissen: Conflict Analysis Across Heterogeneous Viewpoints: Formalization and Visualization
1996-17	Manfred A. Jeusfeld, Tung X. Bui: Decision Support Components on the Internet
1996-18	Manfred A. Jeusfeld, Mike Papazoglou: Information Brokering: Design, Search and Transformation
1996-19 *	P.Peters, M.Jarke: Simulating the impact of information flows in networked organizations
1996-20	Matthias Jarke, Peter Peters, Manfred A. Jeusfeld: Model-driven planning and design of cooperative information systems

1996-21 * G.de Michelis, E.Dubois, M.Jarke, F.Matthes, J.Mylopoulos, K.Pohl, J.Schmidt, C.Woo, E.Yu: Cooperative information systems: a manifesto

1996-22 * S.Jacobs, M.Gebhardt, S.Kethers, W.Rzasa: Filling HTML forms simultaneously: CoWeb architecture and functionality

1996-23 * M.Gebhardt, S.Jacobs: Conflict Management in Design

1997-01 Michael Hanus, Frank Zartmann (eds.): Jahresbericht 1996

1997-02 Johannes Faassen: Using full parallel Boltzmann Machines for Optimization

1997-03 Andreas Winter, Andy Schürr: Modules and Updatable Graph Views for PROgrammed Graph REwriting Systems

1997-04 Markus Mohnen, Stefan Tobies: Implementing Context Patterns in the Glasgow Haskell Compiler

1997-05 * S.Gruner: Schemakorrespondenzaxiome unterstützen die paargrammatische Spezifikation inkrementeller Integrationswerkzeuge

1997-06 Matthias Nicola, Matthias Jarke: Design and Evaluation of Wireless Health Care Information Systems in Developing Countries

1997-07 Petra Hofstedt: Taskparallele Skelette für irregulär strukturierte Probleme in deklarativen Sprachen

1997-08 Dorothea Blostein, Andy Schürr: Computing with Graphs and Graph Rewriting

1997-09 Carl-Arndt Krapp, Bernhard Westfechtel: Feedback Handling in Dynamic Task Nets

1997-10 Matthias Nicola, Matthias Jarke: Integrating Replication and Communication in Performance Models of Distributed Databases

1997-11 * R. Klamma, P. Peters, M. Jarke: Workflow Support for Failure Management in Federated Organizations

1997-13 Markus Mohnen: Optimising the Memory Management of Higher-Order Functional Programs

1997-14 Roland Baumann: Client/Server Distribution in a Structure-Oriented Database Management System

1997-15 George Botorog: High-Level Parallel Programming and the Efficient Implementation of Numerical Algorithms

1998-01 * Fachgruppe Informatik: Jahresbericht 1997

1998-02 Stefan Gruner, Manfred Nagel, Andy Schürr: Fine-grained and Structure-Oriented Document Integration Tools are Needed for Development Processes

1998-03 Stefan Gruner: Einige Anmerkungen zur graphgrammatischen Spezifikation von Integrationswerkzeugen nach Westfechtel, Janning, Lefering und Schürr

1998-04 * O. Kubitz: Mobile Robots in Dynamic Environments

1998-05 Martin Leucker, Stephan Tobies: Truth - A Verification Platform for Distributed Systems

1998-06 * Matthias Oliver Berger: DECT in the Factory of the Future

1998-07 M. Arnold, M. Erdmann, M. Glinz, P. Haumer, R. Knoll, B. Paech, K. Pohl, J. Ryser, R. Studer, K. Weidenhaupt: Survey on the Scenario Use in Twelve Selected Industrial Projects

1998-09 * Th. Lehmann: Geometrische Ausrichtung medizinischer Bilder am Beispiel intraoraler Radiographien

1998-10 * M. Nicola, M. Jarke: Performance Modeling of Distributed and Replicated Databases

1998-11 * Ansgar Schleicher, Bernhard Westfechtel, Dirk Jäger: Modeling Dynamic Software Processes in UML

1998-12 * W. Appelt, M. Jarke: Interoperable Tools for Cooperation Support using the World Wide Web

1998-13 Klaus Indermark: Semantik rekursiver Funktionsdefinitionen mit Striktheitsinformation

1999-01 * Jahresbericht 1998

1999-02 * F. Huch: Verifcation of Erlang Programs using Abstract Interpretation and Model Checking — Extended Version

1999-03 * R. Gallersdörfer, M. Jarke, M. Nicola: The ADR Replication Manager

1999-04 María Alpuente, Michael Hanus, Salvador Lucas, Germán Vidal: Specialization of Functional Logic Programs Based on Needed Narrowing

1999-05 * W. Thomas (Ed.): DLT 99 - Developments in Language Theory Fourth International Conference

1999-06 * Kai Jakobs, Klaus-Dieter Kleefeld: Informationssysteme für die angewandte historische Geographie

1999-07 Thomas Wilke: CTL+ is exponentially more succinct than CTL

1999-08 Oliver Matz: Dot-Depth and Monadic Quantifier Alternation over Pictures

2000-01 * Jahresbericht 1999

2000-02 Jens Vöge, Marcin Jurdzinski A Discrete Strategy Improvement Algorithm for Solving Parity Games

2000-03 D. Jäger, A. Schleicher, B. Westfechtel: UPGRADE: A Framework for Building Graph-Based Software Engineering Tools

2000-04 Andreas Becks, Stefan Sklorz, Matthias Jarke: Exploring the Semantic Structure of Technical Document Collections: A Cooperative Systems Approach

2000-05 Mareike Schoop: Cooperative Document Management

2000-06 Mareike Schoop, Christoph Quix (eds.): Proceedings of the Fifth International Workshop on the Language-Action Perspective on Communication Modelling

2000-07 *	Markus Mohnen, Pieter Koopman (Eds.): Proceedings of the 12th International Workshop of Functional Languages
2000-08	Thomas Arts, Thomas Noll: Verifying Generic Erlang Client-Server Implementations
2001-01 *	Jahresbericht 2000
2001-02	Benedikt Bollig, Martin Leucker: Deciding LTL over Mazurkiewicz Traces
2001-03	Thierry Cachat: The power of one-letter rational languages
2001-04	Benedikt Bollig, Martin Leucker, Michael Weber: Local Parallel Model Checking for the Alternation Free mu-Calculus
2001-05	Benedikt Bollig, Martin Leucker, Thomas Noll: Regular MSC Languages
2001-06	Achim Blumensath: Prefix-Recognisable Graphs and Monadic Second-Order Logic
2001-07	Martin Grohe, Stefan Wöhrle: An Existential Locality Theorem
2001-08	Mareike Schoop, James Taylor (eds.): Proceedings of the Sixth International Workshop on the Language-Action Perspective on Communication Modelling
2001-09	Thomas Arts, Jürgen Giesl: A collection of examples for termination of term rewriting using dependency pairs
2001-10	Achim Blumensath: Axiomatising Tree-interpretable Structures
2001-11	Klaus Indermark, Thomas Noll (eds.): Kolloquium Programmiersprachen und Grundlagen der Programmierung
2002-01 *	Jahresbericht 2001
2002-02	Jürgen Giesl, Aart Middeldorp: Transformation Techniques for Context-Sensitive Rewrite Systems
2002-03	Benedikt Bollig, Martin Leucker, Thomas Noll: Generalised Regular MSC Languages
2002-04	Jürgen Giesl, Aart Middeldorp: Innermost Termination of Context-Sensitive Rewriting
2002-05	Horst Lichter, Thomas von der Maßen, Thomas Weiler: Modelling Requirements and Architectures for Software Product Lines
2002-06	Henry N. Adorna: 3-Party Message Complexity is Better than 2-Party Ones for Proving Lower Bounds on the Size of Minimal Nondeterministic Finite Automata
2002-07	Jörg Dahmen: Invariant Image Object Recognition using Gaussian Mixture Densities
2002-08	Markus Mohnen: An Open Framework for Data-Flow Analysis in Java
2002-09	Markus Mohnen: Interfaces with Default Implementations in Java
2002-10	Martin Leucker: Logics for Mazurkiewicz traces
2002-11	Jürgen Giesl, Hans Zantema: Liveness in Rewriting
2003-01 *	Jahresbericht 2002

2003-02	Jürgen Giesl, René Thiemann: Size-Change Termination for Term Rewriting
2003-03	Jürgen Giesl, Deepak Kapur: Deciding Inductive Validity of Equations
2003-04	Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke: Improving Dependency Pairs
2003-05	Christof Löding, Philipp Rohde: Solving the Sabotage Game is PSPACE-hard
2003-06	Franz Josef Och: Statistical Machine Translation: From Single-Word Models to Alignment Templates
2003-07	Horst Lichter, Thomas von der Maßen, Alexander Nyßen, Thomas Weiler: Vergleich von Ansätzen zur Feature Modellierung bei der Softwareproduktlinienentwicklung
2003-08	Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke: Mechanizing Dependency Pairs
2004-01 *	Fachgruppe Informatik: Jahresbericht 2003
2004-02	Benedikt Bollig, Martin Leucker: Message-Passing Automata are expressively equivalent to EMSO logic
2004-03	Delia Kesner, Femke van Raamsdonk, Joe Wells (eds.): HOR 2004 – 2nd International Workshop on Higher-Order Rewriting
2004-04	Slim Abdennadher, Christophe Ringeissen (eds.): RULE 04 – Fifth International Workshop on Rule-Based Programming
2004-05	Herbert Kuchen (ed.): WFLP 04 – 13th International Workshop on Functional and (Constraint) Logic Programming
2004-06	Sergio Antoy, Yoshihito Toyama (eds.): WRS 04 – 4th International Workshop on Reduction Strategies in Rewriting and Programming
2004-07	Michael Codish, Aart Middeldorp (eds.): WST 04 – 7th International Workshop on Termination
2004-08	Klaus Indermark, Thomas Noll: Algebraic Correctness Proofs for Compiling Recursive Function Definitions with Strictness Information
2004-09	Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Parameterized Power Domination Complexity
2004-10	Zinaida Benenson, Felix C. Gärtner, Dogan Kesdogan: Secure Multi-Party Computation with Security Modules
2005-01 *	Fachgruppe Informatik: Jahresbericht 2004
2005-02	Maximillian Dornseif, Felix C. Gärtner, Thorsten Holz, Martin Mink: An Offensive Approach to Teaching Information Security: "Aachen Summer School Applied IT Security"
2005-03	Jürgen Giesl, René Thiemann, Peter Schneider-Kamp: Proving and Disproving Termination of Higher-Order Functions
2005-04	Daniel Mölle, Stefan Richter, Peter Rossmanith: A Faster Algorithm for the Steiner Tree Problem

2005-05 Fabien Pouget, Thorsten Holz: A Pointillist Approach for Comparing Honeypots

2005-06 Simon Fischer, Berthold Vöcking: Adaptive Routing with Stale Information

2005-07 Felix C. Freiling, Thorsten Holz, Georg Wicherski: Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks

2005-08 Joachim Kneis, Peter Rossmanith: A New Satisfiability Algorithm With Applications To Max-Cut

2005-09 Klaus Kursawe, Felix C. Freiling: Byzantine Fault Tolerance on General Hybrid Adversary Structures

2005-10 Benedikt Bollig: Automata and Logics for Message Sequence Charts

2005-11 Simon Fischer, Berthold Vöcking: A Counterexample to the Fully Mixed Nash Equilibrium Conjecture

2005-12 Neeraj Mittal, Felix Freiling, S. Venkatesan, Lucia Draque Penso: Efficient Reductions for Wait-Free Termination Detection in Faulty Distributed Systems

2005-13 Carole Delporte-Gallet, Hugues Fauconnier, Felix C. Freiling: Revisiting Failure Detection and Consensus in Omission Failure Environments

2005-14 Felix C. Freiling, Sukumar Ghosh: Code Stabilization

2005-15 Uwe Naumann: The Complexity of Derivative Computation

2005-16 Uwe Naumann: Syntax-Directed Derivative Code (Part I: Tangent-Linear Code)

2005-17 Uwe Naumann: Syntax-directed Derivative Code (Part II: Intraprocedural Adjoint Code)

2005-18 Thomas von der Maßen, Klaus Müller, John MacGregor, Eva Geisberger, Jörg Dörr, Frank Houdek, Harbhajan Singh, Holger Wußmann, Hans-Veit Bacher, Barbara Paech: Einsatz von Features im Software-Entwicklungsprozess - Abschlußbericht des GI-Arbeitskreises "Features"

2005-19 Uwe Naumann, Andre Vehreschild: Tangent-Linear Code by Augmented LL-Parsers

2005-20 Felix C. Freiling, Martin Mink: Bericht über den Workshop zur Ausbildung im Bereich IT-Sicherheit Hochschulausbildung, berufliche Weiterbildung, Zertifizierung von Ausbildungsangeboten am 11. und 12. August 2005 in Köln organisiert von RWTH Aachen in Kooperation mit BITKOM, BSI, DLR und Gesellschaft fuer Informatik (GI) e.V.

2005-21 Thomas Noll, Stefan Rieger: Optimization of Straight-Line Code Revisited

2005-22 Felix Freiling, Maurice Herlihy, Lucia Draque Penso: Optimal Randomized Fair Exchange with Secret Shared Coins

2005-23 Heiner Ackermann, Alantha Newman, Heiko Röglin, Berthold Vöcking: Decision Making Based on Approximate and Smoothed Pareto Curves

2005-24 Alexander Becher, Zinaida Benenson, Maximillian Dornseif: Tampering with Motes: Real-World Physical Attacks on Wireless Sensor Networks

2006-01 * Fachgruppe Informatik: Jahresbericht 2005

2006-02 Michael Weber: Parallel Algorithms for Verification of Large Systems

2006-03 Michael Maier, Uwe Naumann: Intraprocedural Adjoint Code Generated by the Differentiation-Enabled NAGWare Fortran Compiler

2006-04 Ebadollah Varnik, Uwe Naumann, Andrew Lyons: Toward Low Static Memory Jacobian Accumulation

2006-05 Uwe Naumann, Jean Utke, Patrick Heimbach, Chris Hill, Derya Ozyurt, Carl Wunsch, Mike Fagan, Nathan Tallent, Michelle Strout: Adjoint Code by Source Transformation with OpenAD/F

2006-06 Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Divide-and-Color

2006-07 Thomas Colcombet, Christof Löding: Transforming structures by set interpretations

2006-08 Uwe Naumann, Yuxiao Hu: Optimal Vertex Elimination in Single-Expression-Use Graphs

2006-09 Tingting Han, Joost-Pieter Katoen: Counterexamples in Probabilistic Model Checking

2006-10 Mesut Günes, Alexander Zimmermann, Martin Wenig, Jan Ritzerfeld, Ulrich Meis: From Simulations to Testbeds - Architecture of the Hybrid MCG-Mesh Testbed

2006-11 Bastian Schlich, Michael Rohrbach, Michael Weber, Stefan Kowalewski: Model Checking Software for Microcontrollers

2006-12 Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker: Replaying Play in and Play out: Synthesis of Design Models from Scenarios by Learning

2006-13 Wong Karianto, Christof Löding: Unranked Tree Automata with Sibling Equalities and Disequalities

2006-14 Danilo Beuche, Andreas Birk, Heinrich Dreier, Andreas Fleischmann, Heidi Galle, Gerald Heller, Dirk Janzen, Isabel John, Ramin Tavakoli Kolagari, Thomas von der Maßen, Andreas Wolfram: Report of the GI Work Group "Requirements Management Tools for Product Line Engineering"

2006-15 Sebastian Ullrich, Jakob T. Valvoda, Torsten Kuhlen: Utilizing optical sensors from mice for new input devices

2006-16 Rafael Ballagas, Jan Borchers: Selexels: a Conceptual Framework for Pointing Devices with Low Expressiveness

2006-17 Eric Lee, Henning Kiel, Jan Borchers: Scrolling Through Time: Improving Interfaces for Searching and Navigating Continuous Audio Timelines

2007-01 * Fachgruppe Informatik: Jahresbericht 2006

2007-02 Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl: SAT Solving for Termination Analysis with Polynomial Interpretations

2007-03 Jürgen Giesl, René Thiemann, Stephan Swiderski, and Peter Schneider-Kamp: Proving Termination by Bounded Increase

2007-04 Jan Buchholz, Eric Lee, Jonathan Klein, and Jan Borchers: coJIVE: A System to Support Collaborative Jazz Improvisation

2007-05 Uwe Naumann: On Optimal DAG Reversal

2007-06 Joost-Pieter Katoen, Thomas Noll, and Stefan Rieger: Verifying Concurrent List-Manipulating Programs by LTL Model Checking

2007-07 Alexander Nyßen, Horst Lichter: MeDUSA - MethoD for UML2-based Design of Embedded Software Applications

2007-08 Falk Salewski and Stefan Kowalewski: Achieving Highly Reliable Embedded Software: An empirical evaluation of different approaches

2007-09 Tina Kraußer, Heiko Mantel, and Henning Sudbrock: A Probabilistic Justification of the Combining Calculus under the Uniform Scheduler Assumption

2007-10 Martin Neuhäußer, Joost-Pieter Katoen: Bisimulation and Logical Preservation for Continuous-Time Markov Decision Processes

2007-11 Klaus Wehrle (editor): 6. Fachgespräch Sensornetzwerke

2007-12 Uwe Naumann: An L-Attributed Grammar for Adjoint Code

2007-13 Uwe Naumann, Michael Maier, Jan Riehme, and Bruce Christianson: Second-Order Adjoints by Source Code Manipulation of Numerical Programs

2007-14 Jean Utke, Uwe Naumann, Mike Fagan, Nathan Tallent, Michelle Strout, Patrick Heimbach, Chris Hill, and Carl Wunsch: OpenAD/F: A Modular, Open-Source Tool for Automatic Differentiation of Fortran Codes

2007-15 Volker Stolz: Temporal assertions for sequential and concurrent programs

2007-16 Sadeq Ali Makram, Mesut Güneç, Martin Wenig, Alexander Zimmermann: Adaptive Channel Assignment to Support QoS and Load Balancing for Wireless Mesh Networks

2007-17 René Thiemann: The DP Framework for Proving Termination of Term Rewriting

2007-18 Uwe Naumann: Call Tree Reversal is NP-Complete

2007-19 Jan Riehme, Andrea Walther, Jörg Stiller, Uwe Naumann: Adjoints for Time-Dependent Optimal Control

2007-20	Joost-Pieter Katoen, Daniel Klink, Martin Leucker, and Verena Wolf: Three-Valued Abstraction for Probabilistic Systems
2007-21	Tingting Han, Joost-Pieter Katoen, and Alexandru Mereacre: Compositional Modeling and Minimization of Time-Inhomogeneous Markov Chains
2007-22	Heiner Ackermann, Paul W. Goldberg, Vahab S. Mirrokni, Heiko Röglin, and Berthold Vöcking: Uncoordinated Two-Sided Markets
2008-01 *	Fachgruppe Informatik: Jahresbericht 2007/2008
2008-02	Henrik Bohnenkamp, Marielle Stoelinga: Quantitative Testing
2008-03	Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, Harald Zankl: Maximal Termination
2008-04	Uwe Naumann, Jan Riehme: Sensitivity Analysis in Sisyphe with the AD-Enabled NAGWare Fortran Compiler
2008-05	Frank G. Radmacher: An Automata Theoretic Approach to the Theory of Rational Tree Relations
2008-06	Uwe Naumann, Laurent Hascoet, Chris Hill, Paul Hovland, Jan Riehme, Jean Utke: A Framework for Proving Correctness of Adjoint Message Passing Programs
2008-07	Alexander Nyßen, Horst Lichter: The MeDUSA Reference Manual, Second Edition
2008-08	George B. Mertzios, Stavros D. Nikolopoulos: The λ-cluster Problem on Parameterized Interval Graphs
2008-09	George B. Mertzios, Walter Unger: An optimal algorithm for the k-fixed-endpoint path cover on proper interval graphs
2008-10	George B. Mertzios, Walter Unger: Preemptive Scheduling of Equal-Length Jobs in Polynomial Time
2008-11	George B. Mertzios: Fast Convergence of Routing Games with Splittable Flows
2008-12	Joost-Pieter Katoen, Daniel Klink, Martin Leucker, Verena Wolf: Abstraction for stochastic systems by Erlang's method of stages
2008-13	Beatriz Alarcón, Fabian Emmes, Carsten Fuhs, Jürgen Giesl, Raúl Gutiérrez, Salvador Lucas, Peter Schneider-Kamp, René Thiemann: Improving Context-Sensitive Dependency Pairs
2008-14	Bastian Schlich: Model Checking of Software for Microcontrollers
2008-15	Joachim Kneis, Alexander Langer, Peter Rossmanith: A New Algorithm for Finding Trees with Many Leaves
2008-16	Hendrik vom Lehn, Elias Weingärtner and Klaus Wehrle: Comparing recent network simulators: A performance evaluation study
2008-17	Peter Schneider-Kamp: Static Termination Analysis for Prolog using Term Rewriting and SAT Solving

2008-18	Falk Salewski: Empirical Evaluations of Safety-Critical Embedded Systems
2008-19	Dirk Wilking: Empirical Studies for the Application of Agile Methods to Embedded Systems
2009-01 *	Fachgruppe Informatik: Jahresbericht 2009
2009-02	Taolue Chen, Tingting Han, Joost-Pieter Katoen, Alexandru Mereacre: Quantitative Model Checking of Continuous-Time Markov Chains Against Timed Automata Specifications
2009-03	Alexander Nyßen: Model-Based Construction of Embedded & Real-Time Software - A Methodology for Small Devices
2009-05	George B. Mertzios, Ignasi Sau, Shmuel Zaks: A New Intersection Model and Improved Algorithms for Tolerance Graphs
2009-06	George B. Mertzios, Ignasi Sau, Shmuel Zaks: The Recognition of Tolerance and Bounded Tolerance Graphs is NP-complete
2009-07	Joachim Kneis, Alexander Langer, Peter Rossmanith: Derandomizing Non-uniform Color-Coding I
2009-08	Joachim Kneis, Alexander Langer: Satellites and Mirrors for Solving Independent Set on Sparse Graphs
2009-09	Michael Nett: Implementation of an Automated Proof for an Algorithm Solving the Maximum Independent Set Problem
2009-10	Felix Reidl, Fernando Sánchez Villaamil: Automatic Verification of the Correctness of the Upper Bound of a Maximum Independent Set Algorithm
2009-11	Kyriaki Ioannidou, George B. Mertzios, Stavros D. Nikolopoulos: The Longest Path Problem is Polynomial on Interval Graphs
2009-12	Martin Neuhäußer, Lijun Zhang: Time-Bounded Reachability in Continuous-Time Markov Decision Processes
2009-13	Martin Zimmermann: Time-optimal Winning Strategies for Poset Games
2009-14	Ralf Huuck, Gerwin Klein, Bastian Schlich (eds.): Doctoral Symposium on Systems Software Verification (DS SSV'09)
2009-15	Joost-Pieter Katoen, Daniel Klink, Martin Neuhäußer: Compositional Abstraction for Stochastic Systems
2009-16	George B. Mertzios, Derek G. Corneil: Vertex Splitting and the Recognition of Trapezoid Graphs
2009-17	Carsten Kern: Learning Communicating and Nondeterministic Automata
2009-18	Paul Hänsch, Michaela Slaats, Wolfgang Thomas: Parametrized Regular Infinite Games and Higher-Order Pushdown Strategies
2010-01 *	Fachgruppe Informatik: Jahresbericht 2010

2010-02	Daniel Neider, Christof Löding: Learning Visibly One-Counter Automata in Polynomial Time
2010-03	Holger Krahn: MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering
2010-04	René Wörzberger: Management dynamischer Geschäftsprozesse auf Basis statischer Prozessmanagementsysteme
2010-05	Daniel Retkowitz: Softwareunterstützung für adaptive eHome-Systeme
2010-06	Taolue Chen, Tingting Han, Joost-Pieter Katoen, Alexandru Mereacre: Computing maximum reachability probabilities in Markovian timed automata
2010-07	George B. Mertzios: A New Intersection Model for Multitolerance Graphs, Hierarchy, and Efficient Algorithms
2010-08	Carsten Otto, Marc Brockschmidt, Christian von Essen, Jürgen Giesl: Automated Termination Analysis of Java Bytecode by Term Rewriting
2010-09	George B. Mertzios, Shmuel Zaks: The Structure of the Intersection of Tolerance and Cocomparability Graphs
2010-10	Peter Schneider-Kamp, Jürgen Giesl, Thomas Ströder, Alexander Serebrenik, René Thiemann: Automated Termination Analysis for Logic Programs with Cut
2010-11	Martin Zimmermann: Parametric LTL Games
2010-12	Thomas Ströder, Peter Schneider-Kamp, Jürgen Giesl: Dependency Triples for Improving Termination Analysis of Logic Programs with Cut
2010-13	Ashraf Armoush: Design Patterns for Safety-Critical Embedded Systems
2010-14	Michael Codish, Carsten Fuhs, Jürgen Giesl, Peter Schneider-Kamp: Lazy Abstraction for Size-Change Termination
2010-15	Marc Brockschmidt, Carsten Otto, Christian von Essen, Jürgen Giesl: Termination Graphs for Java Bytecode
2010-16	Christian Berger: Automating Acceptance Tests for Sensor- and Actuator-based Systems on the Example of Autonomous Vehicles
2010-17	Hans Grönniger: Systemmodell-basierte Definition objektbasierter Modellierungssprachen mit semantischen Variationspunkten
2010-18	Ibrahim Armaç: Personalisierte eHomes: Mobilität, Privatsphäre und Sicherheit
2010-19	Felix Reidl: Experimental Evaluation of an Independent Set Algorithm
2010-20	Wladimir Fridman, Christof Löding, Martin Zimmermann: Degrees of Lookahead in Context-free Infinite Games
2011-01 *	Fachgruppe Informatik: Jahresbericht 2011
2011-02	Marc Brockschmidt, Carsten Otto, Jürgen Giesl: Modular Termination Proofs of Recursive Java Bytecode Programs by Term Rewriting
2011-03	Lars Noschinski, Fabian Emmes, Jürgen Giesl: A Dependency Pair Framework for Innermost Complexity Analysis of Term Rewrite Systems

2011-04 Christina Jansen, Jonathan Heinen, Joost-Pieter Katoen, Thomas Noll: A Local Greibach Normal Form for Hyperedge Replacement Grammars

2011-06 Johannes Lotz, Klaus Leppkes, and Uwe Naumann: dco/c++ - Derivative Code by Overloading in C++

2011-07 Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe: An Operational Semantics for Activity Diagrams using SMV

2011-08 Thomas Ströder, Fabian Emmes, Peter Schneider-Kamp, Jürgen Giesl, Carsten Fuhs: A Linear Operational Semantics for Termination and Complexity Analysis of ISO Prolog

2011-09 Markus Beckers, Johannes Lotz, Viktor Mosenkis, Uwe Naumann (Editors): Fifth SIAM Workshop on Combinatorial Scientific Computing

2011-10 Markus Beckers, Viktor Mosenkis, Michael Maier, Uwe Naumann: Adjoint Subgradient Calculation for McCormick Relaxations

2011-11 Nils Jansen, Erika Ábrahám, Jens Katelaan, Ralf Wimmer, Joost-Pieter Katoen, Bernd Becker: Hierarchical Counterexamples for Discrete-Time Markov Chains

2011-12 Ingo Felscher, Wolfgang Thomas: On Compositional Failure Detection in Structured Transition Systems

2011-13 Michael Förster, Uwe Naumann, Jean Utke: Toward Adjoint OpenMP

2011-14 Daniel Neider, Roman Rabinovich, Martin Zimmermann: Solving Muller Games via Safety Games

2011-16 Niloofar Safiran, Uwe Naumann: Toward Adjoint OpenFOAM

2011-17 Carsten Fuhs: SAT Encodings: From Constraint-Based Termination Analysis to Circuit Synthesis

2011-18 Kamal Barakat: Introducing Timers to pi-Calculus

2011-19 Marc Brockschmidt, Thomas Ströder, Carsten Otto, Jürgen Giesl: Automated Detection of Non-Termination and NullPointerExceptions for Java Bytecode

2011-24 Callum Corbett, Uwe Naumann, Alexander Mitsos: Demonstration of a Branch-and-Bound Algorithm for Global Optimization using McCormick Relaxations

2011-25 Callum Corbett, Michael Maier, Markus Beckers, Uwe Naumann, Amin Ghobeity, Alexander Mitsos: Compiler-Generated Subgradient Code for McCormick Relaxations

2011-26 Hongfei Fu: The Complexity of Deciding a Behavioural Pseudometric on Probabilistic Automata

2012-01 Fachgruppe Informatik: Annual Report 2012

2012-02 Thomas Heer: Controlling Development Processes

2012-03 Arne Haber, Jan Oliver Ringert, Bernhard Rumpe: MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems

2012-04 Marcus Gelderie: Strategy Machines and their Complexity

2012-05 Thomas Ströder, Fabian Emmes, Jürgen Giesl, Peter Schneider-Kamp, and Carsten Fuhs: Automated Complexity Analysis for Prolog by Term Rewriting

2012-06 Marc Brockschmidt, Richard Musiol, Carsten Otto, Jürgen Giesl: Automated Termination Proofs for Java Programs with Cyclic Data

2012-07 André Egners, Björn Marschollek, and Ulrike Meyer: Hackers in Your Pocket: A Survey of Smartphone Security Across Platforms

2012-08 Hongfei Fu: Computing Game Metrics on Markov Decision Processes

2012-09 Dennis Guck, Tingting Han, Joost-Pieter Katoen, and Martin R. Neuhäußer: Quantitative Timed Analysis of Interactive Markov Chains

2012-10 Uwe Naumann and Johannes Lotz: Algorithmic Differentiation of Numerical Methods: Tangent-Linear and Adjoint Direct Solvers for Systems of Linear Equations

2012-12 Jürgen Giesl, Thomas Ströder, Peter Schneider-Kamp, Fabian Emmes, and Carsten Fuhs: Symbolic Evaluation Graphs and Term Rewriting — A General Methodology for Analyzing Logic Programs

2012-15 Uwe Naumann, Johannes Lotz, Klaus Leppkes, and Markus Towara: Algorithmic Differentiation of Numerical Methods: Tangent-Linear and Adjoint Solvers for Systems of Nonlinear Equations

2012-16 Georg Neugebauer and Ulrike Meyer: SMC-MuSe: A Framework for Secure Multi-Party Computation on MultiSets

2012-17 Viet Yen Nguyen: Trustworthy Spacecraft Design Using Formal Methods

2013-01 * Fachgruppe Informatik: Annual Report 2013

2013-02 Michael Reke: Modellbasierte Entwicklung automobiler Steuerungssysteme in Klein- und mittelständischen Unternehmen

2013-03 Markus Towara and Uwe Naumann: A Discrete Adjoint Model for OpenFOAM

2013-04 Max Sagebaum, Nicolas R. Gauger, Uwe Naumann, Johannes Lotz, and Klaus Leppkes: Algorithmic Differentiation of a Complex C++ Code with Underlying Libraries

2013-05 Andreas Rausch and Marc Sihling: Software & Systems Engineering Essentials 2013

2013-06 Marc Brockschmidt, Byron Cook, and Carsten Fuhs: Better termination proving through cooperation

2013-07 André Stollenwerk: Ein modellbasiertes Sicherheitskonzept für die extrakorporale Lungenunterstützung

2013-08 Sebastian Junges, Ulrich Loup, Florian Corzilius and Erika Ábrahám: On Gröbner Bases in the Context of Satisfiability-Modulo-Theories Solving over the Real Numbers

2013-10	Joost-Pieter Katoen, Thomas Noll, Thomas Santen, Dirk Seifert, and Hao Wu: Performance Analysis of Computing Servers using Stochastic Petri Nets and Markov Automata
2013-12	Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl: Alternating Runtime and Size Complexity Analysis of Integer Programs
2013-13	Michael Eggert, Roger Häußling, Martin Henze, Lars Hermerschmidt, René Hummen, Daniel Kerpen, Antonio Navarro Pérez, Bernhard Rumpe, Dirk Thißen, and Klaus Wehrle: SensorCloud: Towards the Interdisciplinary Development of a Trustworthy Platform for Globally Interconnected Sensors and Actuators

* These reports are only available as a printed version.
Please contact biblio@informatik.rwth-aachen.de to obtain copies.