Precise Abstract Interpretation of Hardware Designs

Rajdeep Mukherjee

Balliol College

University of Oxford

A dissertation submitted for the degree of

Doctor of Philosophy

Trinity 2018

Dedicated to Ma, Baba, Rima, Dadu, Dida and Baishali

Abstract

This dissertation shows that the bounded property verification of hardware Register Transfer Level (RTL) designs can be efficiently performed by precise abstract interpretation of a software representation of the RTL.

The first part of this dissertation presents a novel framework for RTL verification using native software analyzers. To this end, we first present a translation of the hardware circuit expressed in Verilog RTL into the software in C called the *software netlist*. We then present the application of native software analyzers based on SAT/SMT-based decision procedures as well as abstraction-based techniques such as abstract interpretation for the formal verification of the software netlist design generated from the hardware RTL. In particular, we show that the path-based symbolic execution techniques, commonly used for automatic test case generation in system softwares, are also effective for proving bounded safety as well as detecting bugs in the software netlist designs. Furthermore, by means of experiments, we show that abstract interpretation techniques, commonly used for static program analysis, can also be used for bounded as well as unbounded safety property verification of the software netlist designs. However, the analysis using abstract interpretation shows high degree of imprecision on our benchmarks which is handled by *manually* guiding the analysis with various trace partitioning directives.

The second part of this dissertation presents a new theoretical framework and a practical instantiation for *automatically* refining the precision of abstract interpretation using Conflict Driven Clause Learning (CDCL)-style analysis. The theoretical contribution is the abstract interpretation framework that generalizes CDCL to precise safety verification for automatic transformer refinement called *Abstract Conflict Driven Learning for Safety* (ACDLS). The practical contribution instantiates ACDLS over a template polyhedra abstract domain for bounded safety verification of the software netlist designs. We experimentally show that ACDLS is more efficient than a SAT-based analysis as well as sufficiently more precise than a commercial abstract interpreter.

Research Hypothesis

This dissertation argues the following thesis.

Precise abstract interpretation of hardware Register Transfer Level (RTL) designs can be achieved by CDCL-style relational safety analysis of a software representation of RTL.

We show this claim with the following evidence.

1. We show that software can be used to represent hardware RTL designs for the purposes of formal verification. We report experiments that demonstrate that a circuit given in Verilog RTL can be automatically translated into software in ANSI-C called the *software netlist* for formal analysis.

2. By means of experiments, we show that path-based symbolic execution techniques are effective for bug finding as well as proving bounded safety properties of some arithmetic and control-intensive software netlist designs generated from the RTL. In addition to that, abstract interpretation with manual guidance is effective for bounded as well as unbounded verification of the software netlist designs.

3. We introduce a new abstract interpretation framework for generalizing CDCL to safety checking over template-based abstract domains called *Abstract Conflict Driven Learning for Safety* (ACDLS). Our experiments show that the instantiation of ACDLS over the template polyhedra abstract domain is sufficiently precise on software netlist designs.

In each of these instances, we analyze the impact of the proposed verification methodology for RTL property checking in terms of performance of the verification engine, the scalability of the techniques and precision of the analysis.

Credit

Credit for technical contribution and presentation of this dissertation is due to

MICHAEL TAUTSCHNIG for the joint work on the implementation of the *v2c* translator that is presented in Chapter 3.

SAURABH JOSHI for the collaboration that underlies the implementation of the path-based symbolic execution engine, VERIFOX, and its application to verify software netlist in Chapter 5.

MARTIN BRIAN for providing an implementation of the single-precision Dual-path floating-point adder/subtractor model, that is used for experiments in Section 5.7.2.

JOHN O'LEARY for the collaboration on the firmware development for UART design, that is presented in Section 5.7.3.

LEOPOLD HALLER for the joint work that led to the development of the ACDLS framework presented in Section 6.8 and Section 6.9.

PETER SCHRAMMEL for the joint work on the implementation of the ACDLS framework that led to the contents in Chapter 7, for providing the implementation of the template polyhedra domain that is presented in Section 7.3.1 and the SSA infrastructure that is presented in Section 7.4.

DANIEL KROENING and TOM MELHAM for the technical suggestions and insight into the application of various software verification techniques including abstract interpretation for hardware property checking, as well as detailed corrections and suggestions on our joint published papers and the presentation in this dissertation.

ARMIN BIERE AND ALESSANDRO ABATE for the detailed corrections, comments and suggestions.

Publications

- *Rajdeep Mukherjee*, Mitra Purandare, Raphael Polig, Daniel Kroening. "Formal Techniques for Effective Co-verification of Hardware/Software Co-designs", In proceedings of the Design Automation Conference, DAC, 2017.

- *Rajdeep Mukherjee*, Peter Schrammel, Leopold Haller, Daniel Kroening, Tom Melham. "Lifting CDCL to Template-based Abstract Domains for Program Verification", In proceedings of the Automated Technology for Verification and Analysis, ATVA, 2017.

- *Rajdeep Mukherjee*, Saurabh Joshi, Andreas Griesmayer, Daniel Kroening, Tom Melham. "Equivalence Checking of a Floating-point Unit Against a High-level C Model", In proceedings of the Formal Methods Symposium, FM, 2016.

- *Rajdeep Mukherjee*, Michael Tautschnig, Daniel Kroening - "v2c - A Verilog to C Translator", In Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems, TACAS, 2016.

- *Rajdeep Mukherjee*, Peter Schrammel, Daniel Kroening, Tom Melham - "Unbounded Safety Verification for Hardware using Software Analyzers", In Proceedings of the Design, Automation and Test in Europe, DATE, 2016.

- *Rajdeep Mukherjee*, Daniel Kroening, Tom Melham - "Hardware Verification using Software Analyzers", In Proceedings of the IEEE Annual Symposium of VLSI, ISVLSI, 2015.

- *Rajdeep Mukherjee*, Daniel Kroening, Tom Melham, Mandayam Srivas - "Equivalence Checking Using Trace Partitioning", In Proceedings of the IEEE Annual Symposium of VLSI, ISVLSI, 2015.

Acknowledgements

I would like to thank my supervisors, Professor Daniel Kroening and Professor Tom Melham, for the advice and guidance they have provided throughout my time as a DPhil student at the Department of Computer Science at University of Oxford. Daniel was extremely supportive and always encouraged me to explore new and exciting territories in formal verification. The interactions with him all these years has taught me several things, both technical and non-technical. I was continually amazed by his willingness to hear my ideas and thoughts and to provide his insightful suggestions, which subsequently convinced me to think about a problem from different perspectives. Apart from his technical supervision, Daniel has provided me helpful career advice and suggestions. I feel very fortunate to be his student. Daniel was and remains my best role model for a scientist and a mentor.

I am very grateful to Tom for all his support, scientific advices, and insightful discussions all these years. Tom has given me the freedom to pursue various projects and at the same time guided me to stay on track with the Oxford DPhil timeline. He is instrumental in helping me with the paper write-ups and formalizing several key concepts presented in this dissertation. I also feel very fortunate to be a student of the Balliol college where Tom is a Professor and Fellow. I will forever be thankful to Balliol college and Tom for giving me this opportunity and providing me technical and non-technical support at various stages of my DPhil.

I am thankful to Professor Armin Biere and Professor Alessandro Abate, for accepting to examine my thesis. Their invaluable feedback and suggestions helped me immensely to improve the dissertation. I also thank Peter Schrammel for his help and guidance as well as participating in numerous discussions with me on abstract interpretation and sat solvers all these years. Peter was always available for discussions and his energy to multi-task has always amazed me. Peter introduced me to a practical implementation of an abstract interpreter and helped me immensely to implement some of the key ideas presented in this dissertation.

I feel privileged to be a member of the Department of Computer Science at University of Oxford. I have had the opportunity to work as a teaching assistant at the Computer Science

department during my DPhil studies which allowed me to interact with undergraduates as well as graduates from different colleges as well as work closely with professors from other research groups. I would like to thank Julie Sheppard for her help and support during the various stages of the DPhil study, starting from the day when I first applied to Oxford as an international student until my last day at Oxford. I am greatly indebted to the generous help and support of the Computer Science department.

I am extremely grateful to Semiconductor Research Corporation (SRC) for sponsoring my DPhil studies at University of Oxford. The SRC studentship gave me the opportunity to work on the SRC funded project and collaborate with the industry liaisons, Intel, IBM and Texas Instruments. I would also like to thank Balliol college for the award of the prestigious Jason Hu scholarship. The generous support from SRC studentship and Jason Hu scholarship have helped me immensely to focus on the research activities and successfully complete my DPhil studies.

I am also grateful to Balliol college for highlighting my doctoral research in the college annual magazine for 2016. During my tenure in Balliol college, I worked as MCR IT officer and represented Balliol MCR in inter-college cricket tournaments. I would also like to mention about the Balliol dining hall where I had several interesting discussions with fellow graduates over lunches and dinners. The graduate accommodation complex at Holywell Manor is one of the wonderful places to live and socialize with the fellow Balliolites and I very much enjoyed my stay at Holywell Manor all these years.

I will forever be thankful to the strong support of all my colleagues and friends. I would like to thank Daniel Poetzl for being so supportive, for those long discussions over coffee and food, for accompanying me to those amazing trips including our first trip to the USA for attending SRC annual review meeting, Pascal Kesseli, Dario Cattaruzza, Lihao Liang, Ganesh Narayanaswamy for being very supportive friends and the numerous discussions we had on various topics in Office 325, Vincent Nimal for sharing his own DPhil experiences and motivating me to join MSR India research lab for internship, Saurabh Joshi, Martin Brian, Bjórn Watcher, and Cristina David for the various stimulating discussions on software verification, Michael Tautschnig for introducing me to the CPROVER framework and collaborating closely on the development of the v2c tool.

I am thankful to Leopold Haller and Vijay D'Silva whose insightful work on abstract interpretation perspective of SAT solvers have motivated me to pursue my DPhil research in this direction. I thank Leopold Haller for introducing me to ACDL, answering my numerous questions in email as well through several skype discussions, and collaborating with me on lifting ACDL to other application domains. I also greatly enjoyed various interesting discussion with Marcelo Sousa, David Landsberg, Daniel Neville, César Rodriguez, Sean

Heelen, John Galea, Youcheng Sun, Matt Lewis, Subodh Sharma, Elizabeth Polgreen, Liana Hadarean, Alex Horn and Ashutosh Natraj at various stages of my DPhil. I am thankful to Peter Schrammel, Daniel Poetzl, Matthias Güdemann, Vincent Nimal, Matthias Schlaipfer, Saurabh Joshi and Leopold Haller for proofreading various chapters of my dissertation. Their feedback helped me greatly to improve this dissertation.

During my PhD, I had the opportunity to work as a Research Intern at ARM Cambridge, IBM Research Zurich and Microsoft Research India. I am extremely grateful to Daryl Stewart at ARM, Mitra Purandare at IBM and Akash Lal at MSR for giving me the internship opportunities. The internship experiences gave me an excellent opportunity to learn about industrial practices of formal verification in both hardware and software industry.

The external collaboration and technical interactions with various researchers outside Oxford also helped me to successfully complete the DPhil work. I would like to thank my external collaborators, Antoine Miné for providing insight into the various domain implementation in Astrée static analyzer, Matthias Heizmann for providing insight into Ultimate Automizer tool, Eugene Goldberg for discussions about hardware verification and SAT solvers and John O'Leary for collaborating closely on using HW-CBMC for firmware/hardware co-verification.

Apart from technical collaborations and interactions, I was deeply interested in music and orchestra. I would like to mention the names of my favourite musicians – Yanni, Samvel Yervinyan, Itzhak Perlman, Andrea Bocelli, Pete Seeger and Bob Dylan, whose music have inherently helped me in good and bad times.

Finally, this dissertation would not have been possible without the unconditional love and support of my parents, my sister, and my grandparents. My grandmother could not see the end of my DPhil, and I have always dreamt of presenting my doctorate degree from Oxford to her. I dedicate this success to her. My heartfelt thanks to them for all the sacrifices they have made in order to help me successfully finish my DPhil. I want to express my gratitude for the love and care of Baishali and her family all these years and instilling me with a strong passion for learning and self-belief. Thank you all very much for making this journey memorable.

Rajdeep Mukherjee
Oxford, United Kingdom

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Formal property verification of hardware designs given in Verilog or VHDL at the RTL description is concerned with checking the logic functionality, the power and security features of RTL against a formal specification. The specification captures the design intent and is expressed in a property specification language, such as SystemVerilog Assertions (SVA). The problem above is known as *Assertion Based Verification*.

Folk wisdom in the formal hardware verification community is to synthesize the RTL design into a bit-level netlist. This technique uses propositional satisfiability solvers (SAT) to reason about the correctness of bit-blasted verification conditions generated from the netlist representation using a bounded or unbounded verification algorithm [35, 37, 218]. Most bit-level verification algorithms heavily rely on the SAT solvers to generate invariants or find counterexamples. This approach misses the opportunity to exploit the word-level structure of the RTL designs. Formal hardware verification tools have therefore developed an alternative word-level flow that synthesizes the RTL design into a word-level netlist. This change in the representation has enabled the use of modern solvers for Satisfiability Modulo Theories (SMT) [144] in the back-end of the tools [7, 24, 138, 207]. However, the performance of word-level symbolic execution engines is determined by the level of abstraction of the symbolic expressions and the power of the rewrite engine used by the SMT solvers.

Unlike traditional approaches that synthesize the RTL design into a bit-level netlist [31, 35, 37] or generate an abstract model of the design (say C/C++ ISA or micro-architectural models derived from RTL [48]), in this dissertation we translate hardware models, articulated in Verilog at register transfer level, into a *software netlist*. The generated model is expressed as a software program. The natural software language for our representation is ANSI-C [1]. Of course, the idea of expressing RTL designs in software has been advocated in the

[1] https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html

past, primarily to enable faster simulation; we will mention only [138], which highlights property verification-related benefits in the context of SoC design. But we emphasize that the software models of hardware used in [138] are abstractions of hardware which are usually written *manually* and are expensive to maintain—and often disconnected from a 'golden' RTL design model from which the chip is ultimately realized. By contrast, our software netlist representation is *not* an abstract model but an *exact* representation of the RTL. The techniques proposed in this dissertation aims at a fully automatic property verification of the synthesizable hardware RTL design via translation into the software netlist.

In this dissertation, we show that the software netlist representation of the hardware RTL opens the door to the application and development of verification technologies inspired by program analysis research, as well as satisfiability research, to formal verification of hardware RTL designs. In particular, the first part of this dissertation presents a novel property verification framework for hardware RTL designs in software language using native software analyzers. To this end, we explore software verification technologies that use SAT/SMT decision procedures [20, 54, 190] including path-wise symbolic execution [44], as well as abstraction-based techniques such as abstract interpretation [70] for formal verification of RTL designs. It is worth emphasizing that some of these techniques have never been applied to formal verification of hardware designs.

For example, abstract interpretation is a well-known technique for static program analysis introduced by Cousot and Cousot [69, 70] that uses abstract semantics to infer important properties of programs over a chosen abstract domain. The main idea of abstract interpretation is to characterize the solutions as fixed point and deriving approximate solutions by fixed-point approximation. Static analysis using abstract interpretation is commonly used for detecting runtime errors such as array overflows, null-pointer exception, or divide-by-zero errors etc. in safety critical systems such as avionics softwares or embedded systems. Typical lattices or abstract domains used in abstract interpretation are non-distributive [27, 130], for example numerical abstract domains such as Intervals, Octagons are non-distributive. Suppose a and b together represent the abstract semantics of a program and c represents the set of abstract states that violate the specification. In a non-distributive domain, $(a \sqcup b) \sqcap c$ can be strictly less precise than $(a \sqcap c) \sqcup (b \sqcap c)$. That is, abstract states of non-distributive lattice support meet operation that precisely model conjunctions and join operation that overapproximate disjunction but these lattices lack negation. Abstract interpretation using non-distributive abstract domains are often imprecise. The precision loss due to join operation is often eliminated by enriching the domain or analysis with disjunction. However, such enrichment makes the analysis very expensive since the number of disjunctive cases that needs to be considered grows significantly as the analysis progresses. Hence, analysis using

distributive abstract domains such as Powerset abstraction [70] or Disjunctive completion domain [70] are precise but expensive.

In this dissertation, abstract interpretation is used for property verification of the software netlist designs generated from the RTL designs. In particular, we have shown by means of experiment that the performance of a commercial abstract interpreter, Astrée [76], is comparable to a bit-level hardware model checker, ABC [37], for bounded as well as unbounded verification of RTL designs. In some cases, Astrée is faster than ABC for finding bugs. However, Astrée shows a high degree of imprecision on our benchmarks. The imprecision of Astrée is handled by manually guiding the analysis using various trace partitioning [182] directives that systematically refines the set of traces by performing the abstraction over a partition of the set of traces instead of the set itself (see details in Section 4.7.3).

The second part of this dissertation argues that the precise abstract interpretation of the software netlist designs can be performed using a CDCL-style [167, 196] analysis that can *automatically* partition the traces of the software netlist through decision and learning procedures, for proving correctness. We will call our technique *Abstract Conflict Driven Learning for Safety*, ACDLS. Owing to the lattice-theoretic characterization of the CDCL algorithm [86, 87], these partitions can be inferred for non-distributive abstract lattices that satisfy certain properties. The model search procedure in ACDLS computes an overapproximation of the greatest fixed point using strongest postcondition transformer for finding a counterexample trace. On the other hand, the conflict analysis procedure in ACDLS computes an underapproximation of the least fixed point using weakest precondition transformer for finding the generalized reason for the conflict from the partial safety proof of the safe trace.

Technology transfer between verification communities

Technology transfer between the hardware verification community and the software verification community has demonstrated tremendous success in the application of these technologies across the two communities that also leads to the development of new verification algorithms in both the fields. Techniques such as predicate abstraction [58, 114], IC3 or Property Directed Reachability (PDR) [31], k-induction [192], and interpolation-based approaches [158] have all made their way from the hardware to the software domain. However, until now, path-based symbolic execution and abstract interpretation have been primarily used for software verification.

Figure 1.1 gives the bird's eye view of the hardware and software verification technologies. We classify these techniques into two : *bug finding* and *proof-based*. Bug-finding

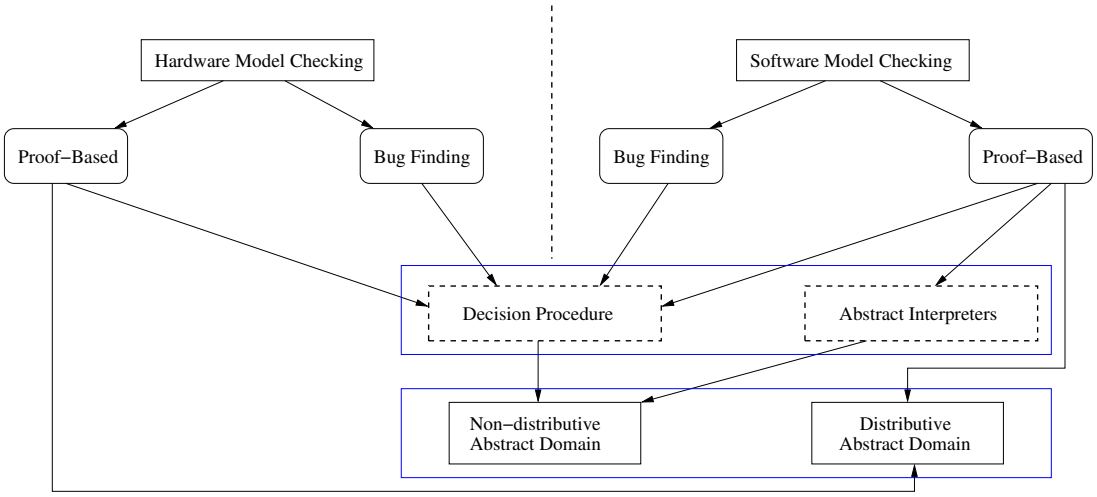


Figure 1.1: Bird's eye view of hardware and software verification technologies

techniques focus on finding behaviors that violate the correctness specification. The bug-finding techniques in software often rely on symbolic execution, which encodes program paths as logical formulas and uses decision procedures to solve them. Bug-finding in hardware uses symbolic simulation to encode a logical representation of the circuit netlist up to a finite unwind bound, which is then solved using a decision procedure – this technique is called Bounded Model Checking (BMC) [22]. Another way to find bugs in a finite-state system is to build an inductive invariant by trying to exclude every trace that breaks induction. By exclusion of a trace, we mean showing that at least one of its states is unreachable. If there is a trace that breaks induction and that cannot be excluded, the property in question does not hold. In contrast, proof-based techniques for hardware and software focus on finding invariants that are sufficient to infer the correctness specification.

Core enabling technology for verification

The core technologies that enable both the bug-finding and proof-based techniques for hardware and software are the *decision procedures* for propositional and first-order logic and *abstract domains* that represent an approximate behavior of the system to be verified.

Classical model checking directly operates on Binary Decision Diagrams (BDD) [42, 61, 66, 92, 221]. Later, model checking techniques based on SAT/SMT decision procedures are proposed [22, 23, 35, 37, 54, 144, 158, 218] for verification of hardware and software systems. In contrast, techniques that combine model checking and symbolic execution, such as Counterexample Guided Abstraction Refinement [59, 62] (CEGAR), used for hardware as well as software verification, operate on abstract domains, known as Powerset abstraction [70], that precisely represent sets of abstract states. Similarly, static program

analysis using abstract interpretation represent the semantics of a program as a set of equations, which are solved iteratively over a chosen abstract domain (which are typically non-distributive domains) using abstract transformers that compute the least fixed point of the set of equations.

At the heart of decision procedures as well as static analyzers lies the abstract domain that they operate on. The choice of the domain determines the precision and the performance of the analysis. We classify these domains into two types – *Distributive* [107], and *non-distributive* [27, 111, 130]. Verification techniques based on Binary Decision Diagrams (BDD) or powerset abstractions use distributive domains. Static analysis using abstract interpretation typically operate on non-distributive domains. SAT-based model checking techniques also operate on non-distributive domains since the SAT solvers themselves operate on a partial assignments domain [177, 196, 226], which are non-distributive structures [86, 87].

Haller et al. [86, 87, 122] makes explicit some fundamental similarities between the way decision procedures and abstract interpretation work (See Section 2.6). They present an understanding of the CDCL architecture in terms of the lattices, fixed points and transformers and concludes that satisfiability solvers are static analyzers [89], which is marked by the top blue box of Figure 1.1. The lattice-theoretic characterization of the CDCL algorithm has several practical applications, namely development of SMT solvers based on abstract interpretation [32, 33] or generalizing CDCL to static program analysis [88, 174]. Hence, the lattice-based characterization of CDCL may be perceived in satisfiability research community as a way to lift the CDCL algorithm to abstract domains (richer than the partial assignments domain) that is suitable for the analysis of the underlying problem instance. In abstract interpretation community, this characterization may be seen as a way to improve the precision of classical abstract interpreter by combining overapproximation of greatest fixed point computation with underapproximation of least fixed point computation and transformer refinement using learning which gives an effect of implicit trace partitioning [182].

Precise Abstract Interpretation of RTL Designs

The work presented in this dissertation combines technologies from software verification as well as satisfiability research for developing novel verification technologies for assertion-based verification of hardware RTL designs. In particular, we present a technique called ACDLS, for precise abstract interpretation of hardware RTL designs using a generalization of the CDCL algorithm over an abstract lattice of traces. Unlike conventional SAT/SMT-based RTL property verification, ACDLS does not perform bit-blasting. Furthermore, as opposed to previous generalizations of CDCL, which focus on Cartesian domains such as Interval

domain to build an Interval-based floating-point decision procedure [32] and a learning-based program analyzer [88] for numerical programs over Intervals, the generalizations presented in this dissertation are unique in several ways. 1) ACDLS supports relational abstract domain, which allows verification of designs that exhibit relational as well as non-relational behaviors. 2) Analysis using ACDLS operates on a logical encoding of the system which is represented by Static Single Assignment formulas. On the other hand, the analysis in [88] is strongly tied to the control-flow graph (CFG). 3) Decision and learning in ACDLS are not tied to control locations, which gives more flexibility for efficient reasoning. By contrast, decision and learning in [88] are strictly restricted to the initial nodes of the CFG. 4) ACDLS is used for bounded property verification of the software netlist design generated from the hardware RTL. By contrast, the work of [88] performs bounds checking on numerical softwares.

Note that all the hardware designs considered in this dissertation are written in Verilog RTL.

A Motivating Example

We give an example to demonstrate the formal property verification of the hardware RTL design using a bit-level hardware model checker, a bit-level software analyzer, a commercial abstract interpreter, and ACDLS.

Figure 1.2 gives a Verilog RTL design (on the left) with two sequential blocks (modeled with always blocks) and a combinational block (modeled with an assign statement). It also contains an initial block that initializes the state-holding elements. The procedural blocks contain non-blocking statements, shown as `(<=)`. Figure 1.4 gives the synthesized hardware (on the left) for the Verilog RTL design of Figure 1.2, while the formal semantics of the RTL is given on the right side of Figure 1.4. The adder circuit and multiplier circuit in the synthesized hardware are shown in vertical boxes, denoted by `ADDER` and `MUL`, respectively. While the registers are shown as square boxes with the name of register marked inside the box and the two Multiplier circuits are denoted by trapezium shapes, with two input lines, one select line and one output line. The formal semantics on the right side of Figure 1.4 contains the semantics of the combinational logic and sequential logic, with respect to the temporal variable t. The Verilog RTL is translated into software netlist in C, as shown on the right side of Figure 1.2. The `main()` procedure of the software netlist is shown on the right side of Figure 1.3, which also contains the initial block. The details of the translation are explained in Chapter 3. We are interested in verifying the correctness of the Verilog RTL design against safety properties.

Verilog RTL	Software netlist
`module top(clk, a, c, out);` `input clk , a;` `output [1:0] c;` `output reg [3:0] out;` `reg b,e;` `reg [1:0] d;` `initial begin` `  b=0;` `  d=2'b0;` `  e=0;` `  out=4'b0;` `end` `assign c = e ? 1'b0 : d;` `always @(posedge clk) begin` `  b<=a;` `  if(b)` `  e <= b;` `  else` `  e <= 0;` `  d <= e + 1;` `end` `always @(posedge clk)` `begin` `  out <= d*d;` `end` `endmodule`	`struct state_elements_top {` `  unsigned int b,e;` `  unsigned char d,out;` `};` `struct state_elements_top u1;` `void top(unsigned int clk, unsigned int a,` `  unsigned char *c,` `  unsigned char *out)` `{` `  // shadow variables` `  unsigned int b_old = u1.b&0x1;` `  unsigned char d_old = u1.d&0x3;` `  unsigned int e_old = u1.e&0x1;` `  u1.b = a;` `  if(b_old)` `  u1.e = b_old&0x1;` `  else` `  u1.e = 0;` `  u1.d = (e_old+1)&0x3;` `  // update the output` `  u1.out = (d_old&0x3) * (d_old&0x3);` `  *out = u1.out&0xF;` `  *c = u1.e ? 0 : (u1.d&0x3);` `}`

Figure 1.2: Circuit to software

Figure 1.3 gives the safety properties that are checked against the RTL design of Figure 1.4. The column on the left of Figure 1.3 gives the SVA properties denoted by the property identifier $P0 \ldots P10$ and the column on the right gives the corresponding assertions in the software netlist. The properties in Figure 1.3 are of two types - global safety properties, and temporal properties containing the implication construct `(|->)` of SVA. The translation of SVA to assertions in the software netlist is described in detail in Section 4.5. Intuitively, the `##N` delay operator in SVA is modeled by invoking the top level procedure `top` in the software netlist N times. Since we are interested in property verification of RTL design by translating them to software netlist, so we consider the notion of equivalence between the Verilog RTL and the software netlist that preserves both the input-output behavior and the validity of all assertions of the RTL in the software netlist. Figure 1.6 gives the alternative ways of modeling the safety properties for the software netlist design of Figure 1.2. The waveform in Figure 1.5 captures the input-output behavior of the RTL design. Using state-of-the-art

System Verilog Assertions

```
P0: assert property (d >= e);
P1: assert property ((a == 1) |-> ##1 (b == 1));
P2: assert property ((a == 1) |-> ##1 (d == 1));
P3: assert property ((a == 1) |-> ##1 (c == 1));
P4: assert property ((a == 1) |-> ##2 (e == 1));
P5: assert property ((a == 0) |-> ##2 (e == 0));
P6: assert property ((a == 1) |-> ##2 (c == 0));
P7: assert property ((a == 1) |-> ##3 (d == 2));
P8: assert property ((a == 0) |-> ##3 (d == 1));
P9: assert property ((a == 0) |-> ##4 (out == 1));
P10: assert property ((a == 1) |-> ##4 (out == 4));
```

Assertions in Software netlist

```
int main()
{
  unsigned int clk, a;
  unsigned char c, out;
  // initial block
  u1.b=0;u1.d=0;u1.e=0;u1.out=0;

  while(1) {
   if(a==1) {
    top(clk,a,&c,&out);
    // Global properies
    P0: assert(u1.d >= u1.e);
    // temporal properties
    P1: assert(u1.b==1);
    P2: assert(u1.d==1);
    P3: assert(c==1);
    top(clk,a,&c,&out);
    P4: assert(u1.e==1);
    P6: assert(c==0);
    top(clk,a,&c,&out);
    P7: assert(u1.d==2);
   }
   else if(a==0) {
    top(clk,a,&c,&out);
    // Global properies
    P0: assert(u1.d >= u1.e);
    top(clk,a,&c,&out);
    // temporal properties
    P5:assert(u1.e==0);
    top(clk,a,&c,&out);
    P8:assert(u1.d==1);
   }
   // check output register
   if(a==1) {
    top(clk,a,&c,&out);
    P10: assert(u1.out==4); }
   else if(a==0)
   {
    top(clk,a,&c,&out);
    P9:assert(u1.out==1); }
  }
}
```

Figure 1.3: Property specification in SVA (on the left) and assertions of the software netlist (on the right)

verifiers, we prove that the result of all assertions in Figure 1.3 are the same in the RTL and the software netlist design. For example, the properties `P2:assert(u1.d==1);` and `P3:assert(c==1);` hold only for the first cycle in both RTL and software netlist, but fails in the subsequent cycles in both the designs. All other properties hold globally in both the designs, that is, these properties are k-inductive for the same value of k in the RTL and the software netlist design.

Table 1.1 gives a comparison between a hardware model checker, two software analyzers and ACDLS for verifying the Verilog RTL design of Figure 1.2 against the specification expressed in SVA, in Figure 1.3. A bit-level hardware model checker, EBMC 4.2 [2], is

[2]http://www.cprover.org/ebmc/

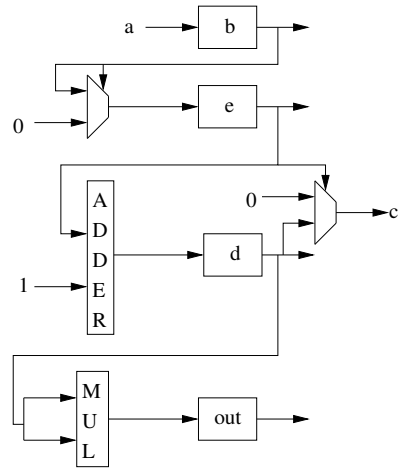
Synthesized Hardware	Formal Semantics
	**Combinational Logic** $\forall t,\ c(t) = if\ e(t)\ then\ 0\ else\ d(t)$ **Sequential Logic** $\forall t,\ b(t+1) = a(t)$ $\forall t,\ e(t+1) = if(b(t))\ then\ b(t)\ else\ 0$ $\forall t,\ d(t+1) = e(t)+1$ $\forall t,\ out(t+1) = d(t)*d(t)$

Figure 1.4: Hardware circuit and its formal semantics

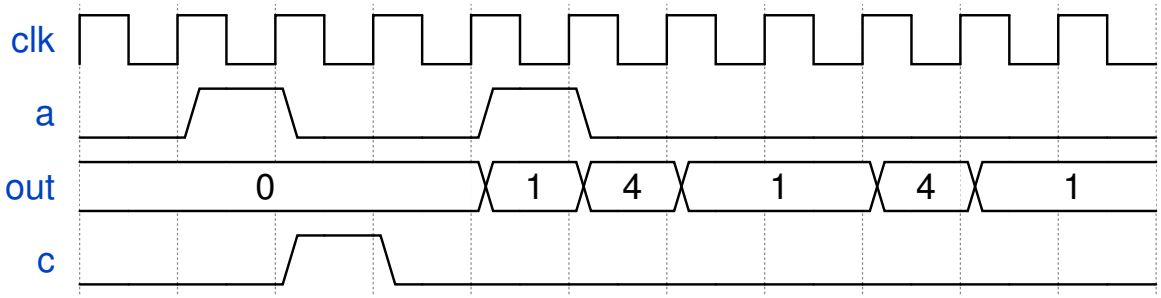


Figure 1.5: Waveform showing the Input-Output behavior of RTL in Figure 1.2

used to verify the Verilog RTL design. EBMC is configured with MiniSAT solver [3] in the backend. Column 2 in Table 1.1 gives the underlying abstract domains used by the hardare and software analyzers. For example, a partial assignments domain is represented by $BVars \rightarrow \{t, f, ?\}$, which maps a Boolean variable to *true* (t), *false* (f) or *unknown* $(?)$. The software netlist is checked using an open source software verification tool, 2LS [190], ACDLS, and a commercial abstract interpretation tool, Astrée [76], which is shown in rows 2, 3 and 4 of Table 1.1, respectively. ACDLS is configured with a product of *BVars* and Interval domain. While abstract interpretation using Astrée is configured with a product of several numeric abstract domains, such as Intervals, Integer bitfields, Octagons and Trace partition domain. 2LS is configured with the MiniSAT solver. The safe and unsafe properties are listed in row 4 of Table 1.1.

[3] http://minisat.se/

Verifier	Domain	decisions	propagations	conflicts	conflict literals	restarts
EBMC (MiniSAT)	$BVars \rightarrow \{t, f, ?\}$	13	307	7	26	1
2LS (MiniSAT)	$BVars \rightarrow \{t, f, ?\}$	1	158	2	2	3
ACDLS	$Itvs[NVars] \times BVars$	1	339	2	2	0
Astrée	Product Domain	safe:{P0,P1,P4,P5,P6,P7,P9.P9.P10} unsafe:{P2,P3}				

Table 1.1: Verification using SAT-based BMC, ACDLS and Abstract interpretation for hardware RTL circuit of Figure 1.2

Analysis using SAT-based tools

The maximum temporal depth among all properties $\{\text{P0} \ldots \text{P10}\}$ is 4, so we unroll the hardware RTL and the software netlist for a bound of 5 (one additional bound than the maximum temporal depth) for checking the properties using EBMC and 2LS [190], respectively. The SAT solver statistics corresponding to EBMC and 2LS are reported in Row 1 and Row 2 of Table 1.1.

Analysis using Abstraction interpretation

The analysis using Astrée is summarized in row 4 of Table 1.1. With manual trace partitioning [182], the analysis using Astrée [4] proves all properties correctly. The trace partitioning directives tell Astrée to keep the traces of the software netlist apart until the next merge point (i.e., the end of the loop or a merge directive).

Analysis using ACDLS

ACDLS is a bounded verification tool. The analysis using ACDLS is summarized in row 3 of Table 1.1. Similar to the SAT-based tools, the software netlist is unwound for a bound of 5. Comparing the solver statistics of ACDLS versus bit-level tools, the SAT solver in EBMC and 2LS clearly makes more decisions and restarts than ACDLS. On the other hand, ACDLS makes slightly more propagations than the SAT-based tools. Compared to Astrée, ACDLS does not require manual partitioning or expensive abstract domains such as the trace partitioning domain to prove the properties. Rather, ACDLS proves the properties using Interval domain only. Furthermore, the analysis using ACDLS does not bit-blast the problem to propositional logic unlike EBMC and 2LS. This example demonstrates that ACDLS is more efficient than SAT-based analysis and sufficiently more precise than classical abstract interpretation.

[4] https://www.absint.com/astree/index.htm

Property Modeling 1

```
int main() {
 unsigned int clk, a;
 unsigned char c, out;
 // initialization
 u1.b=0;u1.d=0;
 u1.e=0;u1.out=0;
 int i=0;
 while(1) {
  assert((u1.d&0x3)
     >= (u1.e&0x1));
  top(clk,a,&c,&out);
  i=i+1;
  if(a==1) {
   assert(u1.b==1);
   if(i==2) {
    assert(u1.e==1);
    assert(c==0); }
   if(i==3) {
    assert(u1.d==2); }
   if(i==4) {
    assert(u1.out==4);
    i=0; } }
  else if(a==0) {
   if(i==2) {
    assert(u1.e==0); }
   if(i==3) {
    assert(u1.d==1); }
   if(i==4) {
    assert(u1.out==1);
    i=0;
   }
  }
 }
}
```

Property Modeling 2

```
int main()
{
 unsigned int clk, a;
 unsigned char c, out;
 // initial
 u1.b=0;u1.d=0;u1.e=0;u1.out=0;
 unsigned int i=0,j=0,k=0,l=0,m=0;
 while(1) {
   top(clk,a,&c,&out);
   j=++i;
   top(clk,a,&c,&out);
   k=++i;
   top(clk,a,&c,&out);
   l=++i;
   top(clk,a,&c,&out);
   m=++i;
   assert( ((!(i==j) || (u1.d>=u1.e)) &&
   (!(k==2)||(u1.d>=u1.e)) &&
   (!(l==3)||(u1.d>=u1.e)) && (!(m==4)||(u1.d>=u1.e))) &&
   (!(a==1) || ((!(k==2) || (u1.e==1 && c==0)) &&
   (!(l==3) || (u1.d==2)) && u1.out==4)) &&
   (!(a==0) || ((!(k==2) || (u1.e==0)) &&
   (!(l==3) || (u1.d==1)) && u1.out==1)) );
   i=0;
 }
}
```

Figure 1.6: Modeling the safety properties {P0,P1,P4,P5,P6,P7,P8,P9,P10} of the software netlist design of Figure 1.2

Contributions of the dissertation

We summarize below the main contributions of this dissertation.

1. We present an automatic translation of the hardware circuit in Verilog RTL into software program called software netlist which is expressed in C language. Unlike previous approaches for generating C/C++ abstract models from Verilog RTL, the translation presented in this dissertation is unique in following three ways.

   (a) The translated software is bit-precise but *not* cycle-accurate.
   (b) The purpose of our translation is assertion-based verification and not simulation.
   (c) Formal RTL verification tools have never synthesized a software from RTL circuit for formal property checking.

   The significance of this translation is that it opens the door to the application of various off-the-shelf program analysis tools for hardware verification. Compared to synthesizing a bit-level netlist from the RTL circuit and checking the bit-blasted

(translating bit-vector arithmetic to propositional logic) formula, the software netlist design enables efficient property verification.

2. We present a novel property checking framework for bounded as well as unbounded verification of RTL designs in software language using native software analyzers. The proposed verification framework is unique in the following two ways.
   (a) For the first time, techniques such as abstract interpretation are used for formal hardware verification,
   (b) It allows for direct technology transfer from program analysis research to formal property checking of hardware.

   The significance of the proposed verification flow is two fold – 1) Path-based symbolic execution techniques are effective for proving bounded safety as well as detecting bugs in the software netlist design generated from RTL circuits such as floating-point arithmetic core, a Universal Asynchronous Receiver Transmitter design and a System-on-a-chip design. 2) By means of experiments, we show that the abstract interpretation with manual guidance is effective for bounded as well as unbounded verification of the software netlist designs.

3. We introduce a new abstract interpretation framework for generalizing CDCL to precise safety checking over template-based non-distributive abstract domains called *Abstract Conflict Driven Learning for Safety*. Unlike previous generalizations of CDCL, ACDLS is unique in the following ways.
   (a) It supports relational abstract domain, which allows verification of designs that exhibit relational as well as non-relational behaviors,
   (b) It operates on a logical encoding of the design which gives more flexibility for efficient deduction, decision and learning,
   (c) Unlike classical abstract interpretation, ACDLS does not require *manual* assistance to recover from imprecision.

   We instantiate ACDLS over the template polyhedra abstract domain for bounded property verification. We experimentally evaluate the performance of ACDLS versus a SAT-based model checker CBMC and a commercial abstract interpreter Astrée. We show that ACDLS is faster and more efficient than CBMC as well as sufficiently more precise than Astrée.

Organization of the dissertation

We outline the organization of the chapters of this dissertation.
Chapter 2 introduces background of hardware designs, symbolic simulation of hardware and hardware synthesis. We present formal preliminaries on logic and satisfiability, partial order and lattices, some basic concepts of programs, the theory of abstract interpretation with formal definitions of various concepts that are used in this dissertation followed by an example of abstract interpretation. Chapter 2 also present the brief overview of the lattice-theoretic generalization of the CDCL algorithm proposed by Haller et al. in [86, 87, 122].

Chapter 3 presents an automatic translation of hardware RTL designs in Verilog to a software netlist in C language for the purposes of formal property verification. We present a tool, *v2c*, for this translation and show two case-studies for property checking.

Chapter 4 explores the use of native software analyzers for the verification of hardware RTL designs. This chapter presents an experimental evaluation for formal verification of RTL designs using the bit-level netlist and software netlist. To this end, we compare the well-known bit-level hardware model checker ABC (winner of the Hardware Model Checking Competition, 2015) and EBMC against various software analyzers from the Software Verification Competition (SV-COMP 2016) for safety verification of hardware RTL designs.

Chapter 5 shows that path-based symbolic execution is an effective bounded verification technique over monolithic BMC-based approaches for RTL circuits that allows input-based partitioning. In this chapter, we present an automatic path-based symbolic execution tool, VERIFOX, for bounded property verification of floating-point arithmetic circuits as well as a Universal Asynchronous Receiver Transmitter (UART) and a System-on-a-Chip design.

Chapter 6 presents a new abstract interpretation framework that generalizes CDCL to precise safety checking over a template-based abstract domains along with soundness, completeness and termination proofs.

Chapter 7 presents a practical instantiation of ACDLS over the template polyhedra abstract domain for sound and precise bounded safety verification. This chapter presents a detailed analysis of the performance of ACDLS in comparison with a precise bit-level bounded model checker, CBMC, and a commercial abstract interpretation tool, Astrée.

Chapter 2

Background

In this chapter, we present background on hardware description languages and hardware synthesis. We also introduce formal preliminaries on logic and satisfiability, partial order and lattices, programs, abstract interpretation and abstract conflict driven clause learning framework.

In this dissertation, we denote equality using the symbol "=". The symbol "$\hat{=}$" is used to denote definitions. We denote $\lambda X.Exp$ as a function that maps a free variable $X \in Exp$ to the value of the expression Exp.

2.1 Hardware Description Languages (HDLs)

In industrial practice, hardware designs are described by means of modeling languages. These include languages to describe schematics and netlists at the lowest level. Higher levels of abstraction can be achieved by hardware description languages (HDLs) such as *VHDL* or *Verilog*.

The challenges in encoding models given in hardware descriptions languages into SAT are mostly shared among all model checking techniques for hardware; they affect BDD-based and SAT-based methods alike. Most HDLs have both *simulation semantics* [94, 178, 188] and *synthesis semantics* [112]. Designers heavily rely on simulation and build models with simulation semantics in mind. Simulation semantics are typically based on an *event-queue*, resembling the data structures maintained by event-driven simulators. On the other hand, the synthesis semantics is closer to the actual hardware produced, and may uncover design flaws that go unnoticed during simulation.

```
module counter(clk, count);
  input clk;
  output [2:0] count;
  reg [2:0] count;

  wire cin =
    ~count[0] & ~count[1] & ~count[2];

  initial count = 3'b0;

  always @ (posedge clk) begin
       count[0] <= cin;
       count[1] <= count[0];
       count[2] <= count[1];
  end
endmodule
```

Figure 2.1: Verilog module of a counter

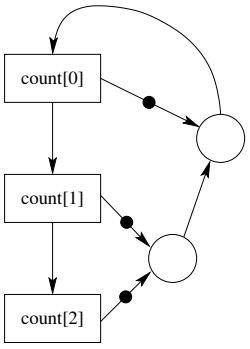


Figure 2.2: Netlist for Figure 2.1

2.1.1 Netlists

A netlist is a collection of primitive elements. A typical way to represent netlists is to use an *And-Inverter Graph* (AIG) [1]. The netlist consists of "and" gates, inverters and memory elements referred to as registers.

Definition 2.1.1. A netlist N is a directed graph (V_N, E_N, τ_N) where V_N is a finite set of vertices, $E_N \subseteq V_N \times V_N$ is the set of directed edges and $\tau_N : V_N \to \{\text{AND}, \text{INV}, \text{REG}, \text{INPUT}\}$ maps a node to its type, where AND is an "and" gate, INV is an inverter, REG is a register, and INPUT is a primary input. The in-degree of a vertex of type AND is at least two, of type INV and REG is exactly one and of type INPUT is zero. Any cycle in N must contain at least one REG node.

As an example, consider the 3-bit counter whose Verilog module is shown in Figure 2.1 (obtained from [50]). The corresponding netlist is shown in Figure 2.2. A node drawn as box represents a REG. A circle-shaped node is an AND gate. An incoming edge of a node marked with a circle indicates negation.

A *state* of a netlist is a mapping of its registers to the Boolean values $\mathbb{B} \hat{=} \{0,1\}$. A netlist N with r registers gives rise to a Kripke structure $M \hat{=} (S_N, I_N, T_N)$ where $S_N = \mathbb{B}^r$ is the set of states and T_N is the transition relation specifying what pairs of states are connected by transitions. The set I_N of *initial states* is determined by the values of the registers immediately after reset. In the above example, $I_N = \neg count[0] \wedge \neg count[1] \wedge \neg count[2]$. Note that S_N for the 3-bit counter in Figure 2.1 consists of $2^3 = 8$ states. An algorithm for obtaining a transition relation for a netlist is given in [60].

[1] http://fmv.jku.at/papers/Biere-FMV-TR-07-1.pdf

The property of a circuit can be given as part of the design description in languages such as *PSL* [2] or as a *System Verilog Assertion* [91].

Hardware Synthesis

Formal verification tools [31, 35, 37] for hardware typically synthesize an input design given in RTL into a netlist. This section briefly describes the bit-level and word-level synthesis process.

Bit-level Synthesis

Logic synthesis [14, 36] is an important step in automatic circuit verification. A network derived from compiling hardware RTL expressed in HDL such as Veriog or VHDL is synthesized into a netlist represented in AIG, before performing technology mapping. The syntheis of RTL into bit-level netlist represented as AIG can be stored in formats such as *AIGER*, *BLIF*, *EDIF*, *PLA* or *BAF*. The netlist captures the effect of one clock period on the state-holding elements. The netlist consists of a network of and-gates, inverters, and memory elements referred to as registers. This approach misses the opportunity to exploit the word-level structure of the input RTL design.

Word-level Synthesis

Word-level reasoning engines have motivated the use of word-level representations for the transition relation [138, 207]. The use of the term "word" refers to a *bit-vector* encoding of the registers and wires, rather than representing them as individual bits. That is, the data-path elements and data packets are treated as words, as opposed to a group of bit-level signals. As in the case of the netlist-based transition relation, the word-level transition relation encodes the effect of one clock period on the state-holding elements. A word-level netlist can be represented in *BTOR* format or some word-level format that resembles SMT-LIB language [3]. This enables the use of word-level decision procedures, such as Satisfiability Modulo Theory (SMT) solvers, in the back-end of these tools. The work of [24] present a word-level model checking framework that uses *transformation-based* approach [108], where a word-level netlist is abstracted to an equivalent but smaller than a gate-level netlist. This is done by rewritting the word-level netlist into a design where the datapath is completely separated from boolean control logic. The word-level registers and input signals are decomposed into a smaller blocks such that the control and data do not overlap. The technique generates a

[2] http://www.eda.org/vfv/docs/PSL-v1.1.pdf
[3] http://smtlib.cs.uiowa.edu/

smaller netlist that can be analyzed using standard gate-level reductions and bit-level model checking algorithms.

2.2 Logic and Satisfiability

In this section, we will describe Propositional logic and Boolean Satisfiability [21, 144, 157, 225]. We discuss the well-known Conflict Driven Clause Learning Boolean satisfiability algorithm and show examples of various learning schemes used in modern day propositional solvers. We will use these formalisms in Chapter 6 and Chapter 7 of this dissertation.

Proposition Logic

Let $Prop$ be the finite set of propositions. The set of propositional formulas over $Prop$ is defined by $\mathcal{F}_{Prop}$. For any proposition $p \in Prop$, $p \in \mathcal{F}_{Prop}$. For any $\phi_1, \phi_2, \ldots, \phi_k \in \mathcal{F}_{Prop}$, their disjunction $\phi_1 \vee \phi_2 \vee \ldots \vee \phi_k \in \mathcal{F}_{Prop}$. Similarly, their conjunction $\phi_1 \wedge \phi_2 \wedge \ldots \wedge \phi_k \in \mathcal{F}_{Prop}$. Further, if $\phi \in \mathcal{F}_{Prop}$, then the negation $\neg\phi \in \mathcal{F}_{Prop}$.

We now define a literal, a clause, and the Conjunctive Normal Form (CNF). For any proposition $p \in Prop$, a literal is a propositional formula of the form, $p \in \mathcal{F}_{Prop}$. A literal is in *positive phase* if it is of the form p and *negative phase* if it is of the form $\neg p$. A clause C is a disjuction of literals, $C = (l_1 \vee l_2 \ldots \vee l_k)$. Clauses are characterized as *unsatisfied*, *satisfied*, *unit* or *unresolved*. A clause is said to be unsatisfied if all the literals in the clause are assigned value 0. A clause is said to be satisfied if at least one of the literals in the clause is assigned value 1. A clause is said to be unit if all literals in the clause but one are assigned value 0. A clause is said to be unresolved if it is neither of the above, that is, neither unsatisfied, nor satisfied and nor unit. A formula ϕ in CNF is a conjunction of clauses, $\phi = C_1 \wedge C_2 \wedge \ldots \wedge C_k$.

Let $\mathbb{B} \hat{=} \{t, f\}$ denote the set of Boolean truth values; where t denotes *true* (1) and f denotes *false* (0). Alternatively, a propositional variable may also be unassigned, which is represented by an *undefined* value u. A propositional assignment is a function $\nu : Prop \rightarrow \{t, f, u\}$, that maps propositions to the Boolean truth values or an undefined value. If all propositions are assigned a value in $\{t, f\}$, then ν is said to as a *complete assignment*. Otherwise, it is a *partial assignment*. Let $Struct$ denote the set of propositional assignments, $Struct \hat{=} Prop \rightarrow \mathbb{B}$. Let $\rho \in Struct$ be a propositional assignment and ϕ be a formula. In this dissertation, we will use the following notation.

$$\rho \models \phi, \text{if } \rho \text{ satisfies } \phi \text{ and } \rho \text{ is called a } \textit{model} \text{ of } \phi$$

$$\rho \not\models \phi, \text{if } \rho \text{ does not satisfy } \phi \text{ and } \rho \text{ is called a } \textit{countermodel} \text{ of } \phi$$

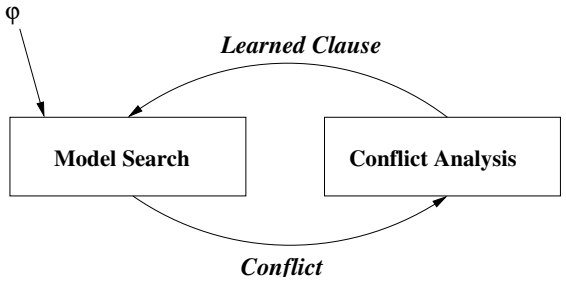


Figure 2.3: Architecture of CDCL

For any proposition $p \in Prop$, the following is true.

1. $\rho \models p$ exactly if $\rho(p) = t$
2. For $\phi = (\phi_1 \vee \phi_2 \vee \ldots \vee \phi_k), \rho \models \phi$ if $\underset{1 \leq i \leq k}{\exists} (\phi_i = t \text{ and } \rho \models \phi_i)$
3. For $\phi = (\phi_1 \wedge \phi_2 \wedge \ldots \wedge \phi_k), \rho \models \phi$ if $\underset{1 \leq i \leq k}{\forall} (\phi_i = t \text{ and } \rho \models \phi_i)$
4. $\rho \models \neg p$ exactly if $\rho(p) = f$

Propositional Satisfiability

A formula ϕ is *satisfiable* if there exists a partial assignment ρ such that $\rho \models \phi$. A formula ϕ is *unsatisfiable* if it has no models. A formula ϕ is *valid* if it has no countermodels.

Given a propositional formula φ, the Boolean Satisfiability (SAT) problem is to determine whether there is a propositional assignment ρ that models each clause in the formula, such that $\rho \models \varphi$. In this chapter, propositional formulas are represented in CNF.

Since the year 2000, we have seen significant advances in satisfiability research, resulting in development of efficient and complete SAT solvers based on the Davis Logemann Loveland (DPLL) procedures [79], local search [191] or Stålmarck's algorithm [193]. Solvers such as *GRASP* [196], *rel_sat* [136], *SATO* [224], *Chaff* [167], use conflict driven clause learning and non-chronological backtracking techniques to effectively prune the search space. We now describe CDCL procedure for solving Boolean Satisfiability problem. The CDCL SAT solvers is heavily inspired by DPLL solvers.

2.2.1 Conflict Driven Clause Learning

The Conflict Driven Clause Learning algorithm [21] is a boolean satisfiability procedure that alternates between the model search phase and conflict analysis phase for solving a propositional formula given in CNF. The architecture of CDCL-based SAT solvers is given in Figure 2.3. The CDCL boolean satisfiability procedure is presented in Algorithm 1 for

Algorithm 1: Conflict Driven Clause Learning Algorithm

```
   input  : A propositional formula φ in CNF and a partial assignment ν
   output : sat or unsat
 1 if deduce(φ,ν)==conflict then
 2 |   return unsat
 3 while ν is not a complete assignment do
   |   /* Decision                                                    */
 4 |   (d,ν) = decide(φ,ν)
 5 |   dlevel ← dlevel + 1
 6 |   ν ← ν ∪ {(d,ν)}
   |   /* Deduction                                                   */
 7 |   if deduce(φ,ν)==conflict then
   |   |   /* Backtrack and Learn                                    */
 8 |   |   blevel = analyzeConflict(φ,ν)
 9 |   |   if blevel < 0 then
10 |   |   |   return unsat
11 |   |   else
   |   |   |   /* Backtracking                                      */
12 |   |   |   backtrack(φ,ν,blevel)
13 |   |   |   dlevel ← blevel
14 end
   /* Found satisfying assignment                                    */
15 return sat
```

a CNF formula φ and an assignment ν that initially maps all propositions to an undefined value. A variable p in CDCL SAT solver has number of properties [21] - the *value*, the *antecedent* and the *decision level*. Recall that a variable can take value from the set {0,1,u}, where *u* denotes an undefined value when a variable is not assigned a value in {0,1}. A variable is *implied* if it is assigned a value through unit clause rule [80] where a unit clause rule identifies the sole unassigned variable in an unit clause and assign it a value 1 for the clause to be satisfied. The antecedent of a variable is the unit clause that is used for implying the variable. The *decision level* $(dlevel)$ of a variable is the depth of the decision tree at which the variable is assigned a value in $\{0,1\}$. The model search phase uses decisions, $decide(\varphi,\nu)$, and deductions, $deduce(\varphi,\nu)$, to search for a satisfying assignment of the formula φ. If φ is falsified at the rool level $(dlevel = 0)$, then the algorithm returns `unsat`. Otherwise, a decision is made. A decision uses branching step to assign a value to a chosen variable. Each branching step can assign two values to a variable, either 0 or 1. Decision is followed by deduction. Deduction uses *Boolean Constraint Propagation* (BCP) [223], which is the repeated application of unit clause rule. If BCP identifies an unsatisfied clause, the CDCL algorithm is said to be in *conflict* and the algorithm enters into conflict analysis phase, $analyzeConflict(\varphi,\nu)$, to identify the reason for the conflict. After a conflict reason is

identified, the algorithm *backjumps* non-chronologically to the backtrack level *blevel* using the procedure, $backtrack(\varphi, \nu, blevel)$. A backjumping procedure undoes all branching steps until the decision level where the solver state is consistent (that is, non-conflicting). This is a key difference with a DPLL solver, which just backtracks to the previous decision level and flips the last decision. The backjumping level in CDCL is determined by analyzing the most recent conflict and learning a new clause from the conflict. If the backjump level is the root of the search tree, then the solver terminates with no models, that is, the formula is *unsatisfiable*, `unsat`. Otherwise, the model search phase is repeated with the learned clause and the procedure continues until all the variables have been assigned, that is ν is a complete assignment. In this case, the algorithm returns `sat` implying that φ is *satisfiable*. The soundness and completeness proofs for different variant of clause learning SAT solvers are presented in [157, 225].

Conflict Analysis in Propositional Satisfiability

The conflict analysis procedure in SAT solver finds the reason for a conflict by analyzing the structure of the unit propagation through *conflict resolution* [21, 144, 157, 225].

The DPLL-based satisfiability procedures use the simplest conflict analysis procedure. It selects a decision variable with the highest decision level when a conflict occurs, undoes all the assignments between this decision level and current decision level and flips the variable if this variable has not been flipped yet. For this, the DPLL algorithm maintains a flag for each decision variable to determine if both phases of the variables has been tried. This procedure leads to *chronological backtracking*.

On the other hand, a CDCL-based SAT solver uses more sophisticated conflict analysis procedure. It relies on implication graph to determine the reasons for conflict. The advantage of conflict analysis procedure in CDCL solvers over DPLL are two folds – 1) it allows non-chronological backjumping, which can discard multiple levels of decisions and deductions stack, 2) it learns a reason for a conflict, called *conflict clause*, and stores them in the clause database to prevent the model search from re-entering into the conflicting search space in future. A conflict clause in a SAT solver is a clause that expresses the fact that some combinations of variable assignment is a countermodel. Clause learning in CDCL-based solvers plays a crucial role in pruning the search-space of the problem when a conflict occurs.

Below, we present some of the well-known conflict analysis procedures used in CDCL-based SAT solvers. Similar to [225, 226], we demonstrate the resolution step by inspecting the *implication graph* [226] for deriving the conflict clause containing the Unique Implication Point (UIP) [167, 196]. An implication graph is a directed acyclic graph (DAG) that records

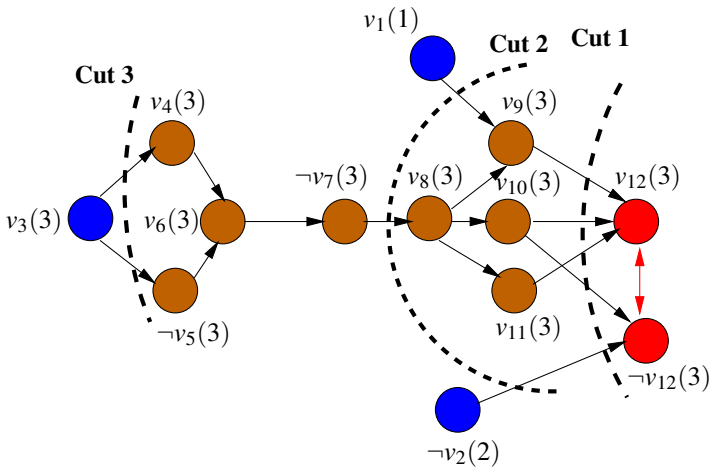


Figure 2.4: Finding UIP in Implication Graph

the implication relationships of variable assignments during the satisfiability solving process. Each vertex in the implication graph represents a variable assignment (positive variable is assigned 1; negative variable is assigned 0). The edges that are incident to each vertex give the reason for that assignment. There is no incident edge for a decision vertex. Each variable is associated with a decision level which denotes the decision at which this variable is assigned, shown by the number within parenthesis corresponding to a variable. A variable is *conflicting* in the implication graph if it is assigned both 0 and 1 at two different vertices. Thus, an implication graph with conflict has two vertices for the conflicting variable. On the other hand, an implication graph with no conflict has at most one vertex for each variable. Figure 2.4 gives an example of implication graph with conflict.

An UIP in the implication graph is a vertex v at the current decision level such that every path from the decision variable of the current decision level to the vertices corresponding to the conflicting variable passes through v. An UIP is a single reason that is sufficient to infer the conflict on the largest decision level. Note that the decision variables are also UIPs. An UIP can be found in an implication graph in linear time [195].

An implication graph is partitioned into two parts – *reason side* containing all decision variables and *conflict side* which contains the conflicting variables. A *cut* is a bipartition of the implication graph such that nodes from the reason side have at least one edge to the conflict side. A conflict clause is constructed from the variable assignments that appears on the reason side of the implication graph. A conflict clause is *asserting* if it contains exactly one UIP of the current decision level. The UIP literal becomes *unit* after backtracking, which makes the conflict clause asserting. The backtracking level is the maximum decision level of the variables that appears in the conflict clause except the variable in the current decision level.

The decision variables in Figure 2.4 are denoted by blue circles. The deduced variables are denoted by brown circles. Red circles are used to denote conflicting variable. The conflicting variable is v_{12} and the conflicting clause is given by $(v_2 \vee \neg v_{10} \vee v_{12})$. The cuts in the implication graph are marked by {cut 1, cut 2, cut3}. Cut 1 of Figure 2.4 gives the conflict clause, $(\neg v_9 \vee \neg v_{10} \vee \neg v_{11} \vee v_2)$. Cut 2 of Figure 2.4 gives the conflict clause, $(\neg v_1 \vee v_7 \vee v_2)$. Cut 3 of Figure 2.4 gives the conflict clause, $(\neg v_3 \vee \neg v_1 \vee v_2)$.

First UIP Learning

The first UIP learning determines the first UIP of the current decision level that is closest to the conflict. The cut corresponding to the first UIP gives the conflict clause in which the variables which are assigned after the first UIP of the current decision level and have paths to the conflicting variable are on the conflict side and the rest variables belongs to the reason side. This leads to an asserting clause since it contains exactly one UIP of the current decision level which will become unit after backtracking. The first UIP based conflict clause of Figure 2.4 is given by $(v_7 \vee \neg v_1 \vee \vee v_2)$.

Last UIP Learning

The cut involving the last UIP cut partitions the implication graph in a way that the variables which are assigned at the current decision level except for the decision variable are kept in the conflict side. The current decision variable and all variable assignments prior to the current decision level are kept on the reason side. This gives an asserting clause that contains the last UIP (current decision variable), which will become unit after backtracking. The last UIP based conflict clause of Figure 2.4 is given by $(\neg v_3 \vee \neg v_1 \vee \vee v_2)$.

All UIP

The All UIP learning partitions the implication graph after the first UIP of each decision level starting from the current decision level up to decision level 1. This corresponds to a learned clause that contains the first UIP of each decision level. A variation of the All UIP learning where the learned clause only contains first UIP of the current decision level is called first UIP learning. Additionally, if the learned clause contains first UIP of immediately previous decision level of the current decision level, it is called 2nd UIP. Similarly, a learned clause can be 3rd UIP and up to All UIP. The key characteristics of this learning scheme is that the learned clause contains exactly one variable at any decision level.

Decision based Learning

The decision based learning procedure partitions the implication graph in a way that the decision variables only appear on the reason side and all the remaining variables are in the conflict side. This corresponds to the conflict clause which contains only those decision variables that are involved in the conflict. A cut of the implication graph corresponding to the decision variables of Figure 2.4 gives the conflict clause $(\neg v_3 \vee \neg v_1 \vee v_2)$. Zhang et al. [225] argues that the pure decision based learning will lead to a weaker learned clause since such combination of decisions can only occur when the solver is restarted again.

Finding Asserting Clause

A conflict clause is made asserting by partitioning the implication graph in a way that keeps one UIP of the current decision level on the reason side and all other variable implied from this UIP on the conflict side. This property of implication graph cutting guarantees that the UIP vertex will become a unit literal after backtracking which makes the conflict clause asserting. The backjumping level in this case is the maximum of the decision level of all variables that appears in the conflict clause except the variable in the current decision level (UIP). An asserting clause of Figure 2.4 is given by $(v_7 \vee \neg v_1 \vee \vee v_2)$, which contains exactly one literal (v_7, first UIP) at the current decision level. The backjumping level is 2 which is derived from the maximum decision level among v_1 and v_2.

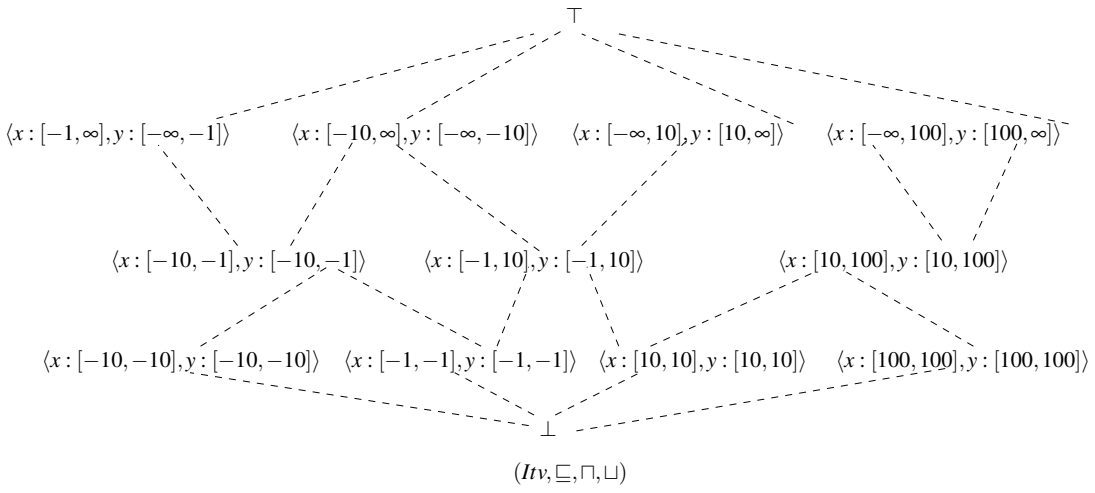


Figure 2.5: Interval lattice over two variables $\{x, y\}$

2.3 Partial Order and Lattices

We now present formal preliminaries related to Partial order and lattices. These concepts are well-known in the literature and are borrowed from [69, 75, 115, 122]. We will use these formalisms for developing the precise abstract interpretation framework in Chapter 6 and Chapter 7 of this dissertation.

Definition 2.3.1. (Set) A *set* S is a collection of objects of any kind. We write $s \in S$ to denote that s is an element of S. We write $A \subset S$ to denote that A is a *subset* of S, that is every element of A is also an element of S. If $A \subset S$ and $A \neq S$, then A is a *proper subset* of S. The *powerset* of the set S, denoted by $\mathbb{P}(S)$, is the set of all subsets of S, including the empty set and S itself. Two sets A and B are *equal*, that is, $A = B$, if they consist of exactly the same elements. Otherwise, we write $A \neq B$. A set that contains no elements is called *empty set* and is denoted by $\emptyset$. The *union* of two sets A and B is denoted by $A \cup B$, is the set of all elements belonging to at least one of the two sets. The *intersection* of two sets A and B denoted by $A \cap B$, consists of all elements belonging to both A and B. Two sets are *disjoint* if $A \cap B = \emptyset$. We write $A \setminus B$ to denote the set of all elements in A that don't belong to B. The *Cartesian product* of two non-empty sets A and B, denoted by $A \times B$, is the set of all ordered pairs (a,b) such that $a \in A$ and $b \in B$. Given a set A, we denote by A^* the set of *finite sequences* over elements of A. We write A^+ to denote the set of finite non-empty sequences over elements of A.

Definition 2.3.2. (Relation) Given two non-empty sets, A and B, a *relation* R from A to B is a subset of the Cartesian product set $A \times B$. This subset is inferred by describing a relationship between the elements of the ordered pairs in $A \times B$.

Definition 2.3.3. (Function) A *function* f from a set A to B, denoted by $f : A \to B$, is a mapping that assigns to each $a \in A$ an unique element $b \in B$ such that $f(a) = b$. Here, A is called the *domain* of f, B is *codomain* and the element $f(a) \in B$ the *image* of a under f. A function $f : A \to B$ is *total* if *every* element $a \in A$ is mapped to an unique element of B, that is, $f(a_1) \neq f(a_2)$ for every pair $(a_1, a_2) \in A$ such that $a_1 \neq a_2$. On the other hand, *partial* functions need not be defined for all the values in its domain. If $f : A \to B$ and $g : B \to C$, the composition $f \circ g$ is given by the function $h : A \to C$ such that $h(a) = g(f(a))$ for every $a \in A$. We write $f[a \mapsto b]$ to denote the function that maps $a \in A$ to $b \in B$ and every $a' \neq a$ to $f(a')$. We denote an identity function over set A by $id_A : A \to A$.

Definition 2.3.4. (Monotonic function). A function f is *monotonic* iff it is either entirely non-increasing or entirely non-decreasing. f is called *monotonically increasing* if for all x,

y such that $x \leq y$, $f(x) \leq f(y)$. f is called *monotonically decreasing* if for all x, y such that $x \leq y$, $f(x) \geq f(y)$.

Definition 2.3.5. (Partial Order) Given a set P, a *partial order* is a binary relation $\leq$ over elements of P that satisfies the following property for all $a, b, c \in P$

1. Reflexivity: $a \leq a$.
2. Anti-symmetric: If $a \leq b$ and $b \leq a$, then $a = b$.
3. Transitive: If $a \leq b$ and $b \leq c$, then $a \leq c$.

Definition 2.3.6. (Poset) A *Partially Ordered Set* or *Poset* $(P, \leq)$ is a set P with the binary relation $\leq$ such that elements of P are partially ordered.

Definition 2.3.7. (Mapping between Posets) For Posets $(P, \leq)$ and $(Q, \subseteq)$, a function $f : P \to Q$ is called *monotonic* or *order preserving* if for elements $p, p' \in P$, $p \leq p' \implies f(p) \subseteq f(p')$.

Definition 2.3.8. (Upwards Closed Set) An upward closed set or *upset* of a partially ordered set $(P, \subseteq)$ is a subset Q such that if $a \in Q$ and $a \subseteq b$, then this implies that $b \in Q$.

Definition 2.3.9. (Downwards Closed Set) A downwards closed set or *downset* of a partially ordered set $(P, \subseteq)$ is a subset Q such that if $a \in Q$ and $b \subseteq a$, then this implies that $b \in Q$.

Definition 2.3.10. (Lattice) A lattice is a poset $(L, \sqsubseteq)$ in which every two elements have a meet denoted by $(\sqcap)$ and join denoted by $(\sqcup)$. The meet precisely models set intersection (or conjunction, taking a logical view), and the join over-approximates set union (or disjunction). In this dissertation, we denote a lattice by $(L, \sqsubseteq)$ or $(L, \sqsubseteq, \sqcup, \sqcap)$. An element $l \in L$ is called *top element* denoted by $\top$, if every element of L is less than l. Dually, an element $l \in L$ is called *bottom element* denoted by $\bot$, if every element of L is greater than l.

Figure 2.5 shows an example of Interval lattice over two integer variables $\{x, y\}$. An element of Interval lattice denoted by, $Itv = (ItvElm, \sqsubseteq_I, \sqcup_I, \sqcap_I)$, is represented by $ItvElm : (Var \to Itv) \cup \{\bot\}$, where Itv is the set of intervals of type $[l, u]$ over numeric data type with $l \leq u$. The least element is $\bot$ and the greatest element is $\top$ which maps all variables to their minimum (*min*) and maximum (*max*) values. An interval $\langle x : [min, v] \rangle$ is written as $x \leq v$. The partial order $\sqsubseteq_I$ over elements in the set Itv is given by $I_1 \sqsubseteq_I I_2$ if I_2 contains I_1.

Definition 2.3.11. (Complete Lattice) A lattice $(L, \sqsubseteq)$ is *complete* in which all subsets $P \subseteq L$ have both a join $\bigsqcup P$ and a meet $\bigsqcap P$.

Definition 2.3.12. (Bounded Lattice) A lattice $(L, \sqsubseteq)$ is called *bounded* if it contains a greatest element *top* ($\top$) and a least element *bottom* ($\bot$).

Definition 2.3.13. (Powerset lattice) Given a set S, a powerset lattice, $(\mathbb{P}(S), \subseteq, \cup, \cap)$, is a complete lattice that is constructed from the powerset of S, $\mathbb{P}(S)$.

Definition 2.3.14. (Distributive Lattice) A lattice $(L, \sqsubseteq)$ is called *distributive* if it satisfies the distributive property given by,

$$\forall p, q, r \in L,\ p \sqcap (q \sqcup r) = (p \sqcap q) \sqcup (p \sqcap r) \text{ and } p \sqcup (q \sqcap r) = (p \sqcup q) \sqcap (p \sqcup r) \text{ holds}$$

A lattice is called *non-distributive* if it does not satisfy the distributive property.

Example 2.3.1. An example of a Distributive lattice is a Powerset lattice. An example of a non-distributive lattice is an Interval lattice. Consider the set of Intervals in an Interval lattice given by $\langle I_1 : [0,2] \rangle$, $\langle I_2 : [8,9] \rangle$, $\langle I_3 : [5,6] \rangle$, then $(I_1 \sqcup I_2) \sqcap I_3 \neq (I_1 \sqcap I_3) \sqcup (I_2 \sqcap I_3)$ since $(I_1 \sqcup I_2) \sqcap I_3 = [5,6]$ and $(I_1 \sqcap I_3) \sqcup (I_2 \sqcap I_3) = \bot$. Similarly, it can be shown that $(I_1 \sqcap I_2) \sqcup I_3 \neq (I_1 \sqcup I_3) \sqcap (I_2 \sqcup I_3)$.

Definition 2.3.15. (Complement in Lattice) Given a bounded lattice $(L, \sqsubseteq)$, and an element $a \in L$, the complement of a is an element $\hat{a} \in L$ such that $a \sqcap \hat{a} = \bot$ or $a \sqcup \hat{a} = \top$. A complementation property of a lattice is a function that maps every element of a lattice to one of its complements.

Theorem 2.3.1. In a bounded distributive lattice, a complement if exists is unique.

Proof. Let $(L, \sqsubseteq, \sqcap, \sqcup)$ be a bounded distributive lattice. Let $a \in L$ and $b, c \in L$ are two complments of a. Then, $b = \top \sqcap b = (a \sqcup c) \sqcap b = (a \sqcap b) \sqcup (c \sqcap b) = \bot \sqcup (c \sqcap b) = (c \sqcap b)$. If we interchange b with c in the above derivation, then it gives that $c = (c \sqcap b)$. Hence, $b = c$, which implies that $a \in L$ has unique complement. □

Definition 2.3.16. (Chain and Anti-chain) A chain in a poset P is a set $C \subseteq P$ such that any two elements in C are comparable. An anti-chain in a poset P is a set $A \subseteq P$ such that *no* two elements in A are comparable.

An increasing sequence of elements of P of the form $a_0 \sqsubseteq a_1 \sqsubseteq a_2 \ldots$ defines a chain. Similarly, a decreasing sequence of elements also defines a chain. A descending chain condition for a poset P is given by any chain $a_0 \sqsupseteq a_1 \sqsupseteq a_2 \ldots$ such that there exists a number n for which $a_n = a_{n+1} = \ldots$, that is, every non-empty set $Q \subseteq P$ has at least one minimal element in P. The ascending chain condition is defined dually.

Definition 2.3.17. (Height of Lattice) The height of a lattice is defined as the maximum cardinality of a chain. A lattice has finite height if it satisfies both the ascending chain condition and descending chain condition. The height of an element $a \in L$ is defined as the longest path from a to $\bot$.

Definition 2.3.18. (Additive and Multiplicative function) A function f on a complete lattice $(L, \sqsubseteq)$ is called *additive* or *join preserving* if $f(a \sqcup b) = f(a) \sqcup f(b)$. Function f is called *completely additive* if $f(\bigsqcup X) = \bigsqcup f(X)$. The dual notions are called *multiplicative* or *meet preserving* and *completely multiplicative*.

Definition 2.3.19. (Idempotent function) A function f on a complete lattice $(L, \sqsubseteq)$ is called *Idempotent* if $f(f(a)) = f(a)$ for all a.

Definition 2.3.20. (Reductive and Extensive function) A function f on a complete lattice $(L, \sqsubseteq)$ is called *reductive* if $f(a) \sqsubseteq a$ for all a. Function f is called *extensive* if $f(a) \sqsupseteq a$ for all a.

2.4 Programs

In this section, we discuss formal preliminaries about programs. For a program P, we denote by *Var* a finite set of program variables and *Val* as the set of values that the variables can take. The computation and control-flow of a program is represented graphically by a control-flow graph, which is defined next.

Definition 2.4.1. (Control-flow Graph). A *control-flow graph* (CFG) is a directed graph with 5 tuples $(V, E, \mathcal{S}, \mathcal{T}, I)$, where V is a set of nodes or control locations, $E \subseteq V \times V$ is a set of control-flow edges, $\mathcal{S}$ is a set of actions modelling *program statements*, $\mathcal{T} : E \to \mathcal{S}$ is a set of labelled transitions which encodes the control-flow, $I \in V$ is the initial location.

Definition 2.4.2. (Memory state). A *memory state*, $\Omega \mathrel{\hat{=}} \mathit{Var} \mapsto \mathit{Val}$ is a mapping of program variables *Var* to its corresponding value *Val*.

Definition 2.4.3. (Program State). A *program state*, $\sigma \mathrel{\hat{=}} V \times \Omega$, is a tuple consisting of control location and memory state.

Definition 2.4.4. (Path). A *path* through a CFG is a finite, non-empty sequence of control-locations $v_1 \dots v_n$ such that $\underset{1 \le i \le n}{\forall i} (v_i, v_{i+1}) \in E$.

Definition 2.4.5. (Concrete Environment) A *concrete environment* can be regarded as memory state that contains values for each variable in the program.

Definition 2.4.6. (Concrete domain) Let *Env* be the set of all possible concrete environments. A *concrete domain* is defined by a complete lattice, $(\mathbb{P}(Env), \subseteq, \cup, \cap)$.

Each statement $s \in \mathcal{S}$ modifies the state of a CFG. Hence, execution of a program statement s corresponds to a state transition in a transition relation given by, $\mathcal{ST}_s \subseteq \Omega \times \Omega$. The operational semantics of a CFG can be defined using a *state transition system*.

Definition 2.4.7. (State Transition System). A *state transition system* of a Control Flow Graph $G = (V, E, \mathcal{S}, \mathcal{T}, I)$ is a 3 tuple $M = (\Sigma, \mathcal{R}, I)$, where Σ is a set of program states $\Sigma = (V \times \Omega)$, $I \subseteq \Sigma$ is the set of initial states given by, $I = \{(I, \omega) \mid \omega \in \Omega\}$, and $\mathcal{R} \subseteq \Sigma \times \Sigma$ is a transition relation defined as $((v_1, \omega_1), (v_1, \omega_2)) \in \mathcal{R}$ where $(v_1, v_2) \in E$ and $(\omega_1, \omega_2) \in \mathcal{ST}_{\mathcal{T}(v_1, v_2)}$.

Example 2.4.1. Consider a program, {`x:=10; while(x>0) x:=x-1;`}. The transition relation over set of integers $\mathbb{Z}$ is given by $\{\langle x, x' \rangle \mid x > 0 \wedge x' = x - 1\}$, where the initial state is $\{10\}$.

Definition 2.4.8. (Trace). A *trace* π of state transition system $(\Sigma, \mathcal{R}, I)$ of a CFG G is a finite sequence of states $\sigma_1 \ldots \sigma_n$ such that $\underset{1 \leq i \leq n}{\forall i}\ \sigma_i \in \Sigma$. We denote a set of traces by Π. A trace $\pi = (v_1, \omega_1) \ldots (v_k, \omega_k) \in \Pi$ is *well formed* if $(v_1, \omega_1) \in I$ and π follows the transition relation $\mathcal{R}$, that is $(v_1, \omega_1) \rightarrow (v_2, \omega_2) \rightarrow \ldots \rightarrow (v_k, \omega_k)$. We denote a set of well-formed traces by Π_{wf}. A prefix of a trace is a contiguous sub-sequence of states with the initial state (I, ω), where $\omega \in \Omega$. A trace of length n is a finite sequence of n states $\sigma_1 \ldots \sigma_n$.

2.5 Abstract Interpretation

Abstract Interpretation [28,67,69,75] is a mathematical theory of sound approximation of the semantics of program. In abstract interpretation, the semantics of the program is represented by a set of equations, which are solved over a chosen abstract domain by computing a fixed point of abstract transformers. We use the terms "precise" and "exact" in this dissertation in the following context. Given two abstract states a and b from an abstract lattice, we say a is more precise than b if the concretization of a gives a smaller set than the concretization of b. On the other hand, an abstract state a is exact with respect to a concrete state c if the concretization of a gives exactly the set c.

In this section, we present formal preliminaries related to abstract interpretation. In this dissertation, we use the Galois connection [72] to formalize the notion of sound approximation. The Galois connection framework is described next.

Definition 2.5.1. (Galois connection) Let $(C, \subseteq)$ be a *concrete lattice* and $(A, \sqsubseteq)$ be an *abstract lattice*. We define a pair of functions (α, γ) such that the *abstraction function* $\alpha\colon C \to A$, maps concrete elements to their best abstract representation, and a *concretization function*, $\gamma\colon A \to C$, maps an abstract element to a set of elements in the concrete domain that are approximated through the corresponding abstraction. Given an identify function id_X on lattice X, the pair (α, γ) is a *Galois connection* if the following properties hold.

$$\alpha \circ \gamma \sqsubseteq id_A \quad \text{and} \quad \gamma \circ \alpha \supseteq id_C$$

Intuitively, this means that (α, γ) is a Galois connection if $\underset{\{a \in A, c \in C\}}{\forall} \alpha(c) \sqsubseteq a$ iff $\gamma(a) \supseteq c$.

Example 2.5.1. An example of Galois connection between a powerset of integers $(\mathbb{P}(\mathbb{Z}), \subseteq)$ and interval lattice $(Itv, \sqsubseteq)$ is given by,

$$(\mathbb{P}(\mathbb{Z}), \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (Itv, \sqsubseteq)$$

$$\alpha(\emptyset) \hat{=} \bot \quad \alpha(z) \hat{=} [min(z), max(z)]$$

$$\gamma(\bot) \hat{=} \emptyset \quad \gamma([a,b]) \hat{=} \{z \in \mathbb{Z} \mid a \leq z \leq b\}$$

Definition 2.5.2. (Properties of Galois connection) A Galois connection between a concrete lattice $(C, \subseteq)$ and an abstract lattice $(A, \sqsubseteq)$, denoted by $(C, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$, has the property that the abstraction function α is completely additive, the concretization function γ is completely multiplicative and the Galois connection can be sequentially composed to generate a series of approximations as shown below,

$$(A, \subseteq) \underset{\alpha_a}{\overset{\gamma_a}{\leftrightarrows}} (B, \sqsubseteq) \underset{\alpha_b}{\overset{\gamma_b}{\leftrightarrows}} (C, \preccurlyeq) \implies (A, \subseteq) \underset{\alpha_a \circ \alpha_b}{\overset{\gamma_a \circ \gamma_b}{\leftrightarrows}} (C, \sqsubseteq)$$

Definition 2.5.3. (Disjunctive Abstract Domain) Let $(A, \sqsubseteq, \sqcup, \sqcap)$ be an abstraction of the concrete domain $(C, \subseteq, \cup, \cap)$ w.r.t. a Galois connection (α, γ). The abstract lattice A is *disjunctive* if for the abstract elements $a, b \in A$, $\gamma(a \sqcup b) = \gamma(a) \cup \gamma(b)$.

Definition 2.5.4. (Exact and Approximate Abstraction) Let concrete semantics C be approximated by an abstract semantics A with an abstraction function $\alpha\colon C \mapsto A$. Then, the abstraction is *exact* if $\alpha(C) = A$ and *approximate* if $\alpha(C) \sqsubset A$.

Definition 2.5.5. (Pointwise lifting) Let $(A, \sqsubseteq, \sqcap, \sqcup)$ be a poset. A *pointwise lifting* is an operation that lifts the pointwise order $\sqsubseteq$ to the functions on A. Consider the functions $f, g\colon T \to A$, where T is a set. The pointwise order $\sqsubseteq$ lifted to f, g is a partial order, denoted by $f \sqsubseteq^{\circ} g$, holds, if $f(a) \sqsubseteq g(a)$ for all a. Similarly, the pointwise meet and pointwise join on A holds if $f(a) \sqcap g(a)$ and $f(a) \sqcup g(a)$, respectively, for all a.

Definition 2.5.6. (Transformer). A *transformer* f on a poset $(P, \sqsubseteq)$ is either monotonically increasing or monotonically decreasing function denoted by, $f\colon P \to P$, that is f is order preserving.

Definition 2.5.7. (Upper closure and Lower Closure Transformer). A transformer f on a poset $(P, \sqsubseteq)$ is *upper-closure* if it is idempotent and extensive. A transformer f is *lower-closure* if it is idempotent and reductive.

Definition 2.5.8. (Approximating Transformer). Let $(P, \subseteq)$ and $(Q, \sqsubseteq)$ be two posets such that $(P, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (Q, \sqsubseteq)$ with transformers $f\colon P \to P$ and $g\colon Q \to Q$. We say transformer g *soundly approximates* transformer f if one of the following condition holds.

1. Let Q overapproximate P through (α, γ), then g *soundly overapproximates* f if $\forall q \in Q, f \circ \gamma(q) \subseteq \gamma \circ g(q)$.
2. Let Q underapproximate P through (α, γ), then g *soundly underapproximates* f if $\forall q \in Q, f \circ \gamma(q) \supseteq \gamma \circ g(q)$.

Definition 2.5.9. (Best Abstract Transformer). Let $(P, \subseteq)$ and $(Q, \sqsubseteq)$ be two posets such that $(P, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (Q, \sqsubseteq)$ with transformers $f\colon P \to P$ and $g\colon Q \to Q$. The *best abstract transformer* of f is $\alpha \circ f \circ \gamma$.

Definition 2.5.10. (Fixed point) Let $(A, \leq)$ be a poset. Then, an element $a \in A$ is a *fixed point* of a function F if $F(a) = a$. The *pre-fixed point* of F is defined as those elements $a \in A$ such that $f(a) \leq a$. The *post-fixed point* of F is defined as those elements $a \in A$ such that $a \leq f(a)$. A fixed point is both a pre-fixed point and a post-fixed point. The *least fixed point* of F, denoted by $\mathsf{lfp}\, F$, is the least among fixed points of F. The *greatest fixed point*, denoted by $\mathsf{gfp}\, F$, is the greatest among fixed points of F. A *least pre-fixed point* of F is the least among pre-fixed points of F. A *greatest post-fixed point* of F is the greatest among post-fixed points of F. The lfp, gfp, pre-fixed points and post-fixed points are unique whenever they exist.

Definition 2.5.11. (Knaster-Tarski Theorem) Let L be a complete lattice and $f\colon L \to L$ be an order-preserving function. Then the set of fixed points in L also forms a complete lattice, where the least fixed point (lfp) and greatest fixed point (gfp) are given as follows.

$$\mathsf{lfp}\, f = \bigsqcap \{a \in L \mid f(a) \sqsubseteq a\}, \quad \mathsf{gfp}\, f = \bigsqcup \{a \in L \mid f(a) \sqsupseteq a\}$$

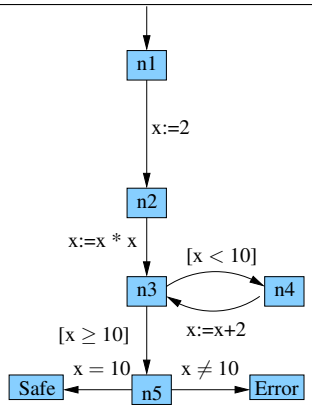
Program	Control-Flow Graph	Static Analysis Equation
`int main() {` `x=2;` `x=x*x;` `while(x<10)` `x=x+2;` `assert(x==10)` `}`		$S_{n1} = \top,$ $S_{n2} = post_{x:=2}(S_{n1}),$ $S_{n3} = post_{x:=x*x}(S_{n2}) \cup post_{x:=x+2}(S_{n4}),$ $S_{n4} = post_{x<10}(S_{n3}),$ $S_{n5} = post_{x\geq 10}(S_{n3})$ $S_{Error} = post_{x\neq 10}(S_{n5})$ $S_{Safe} = post_{x=10}(S_{n5})$

Figure 2.6: A Control-Flow Graph and its static analysis equation

An Example of Abstract Interpretation

We present an example of a classical abstract interpretation of program. Figure 2.6 shows a Control-Flow Graph and its corresponsing static analysis equations [163]. A static analysis equation encodes the data-flow between individual control-flow nodes in the CFG and is given by a set valued variable S_n for each location n in the CFG. Assume that each statement s in the program is associated with a postcondition transformer, $post_s(S_n)$, that computes the successor state of a statement s starting from S_n that can be reached in one step. Static analysis using abstract interpretation usually computes a fixed point over the static analysis equations obtained from the control-flow graph representation of a program [70].

Let us consider the equations in Figure 2.6 that models the loop, $S_{n3} = post_{x:=x*x}(S_{n2}) \cup post_{x:=x+2}(S_{n4})$, $S_{n4} = post_{x<10}(S_{n3})$. These equations can be written as a function, $F(X) = \{4\} \cup \{x+2 \mid x \in X, x \leq 10\}$, where $post_{x:=x*x}(S_{n2}) = 4$. Assuming variable x is an integer, the lattice of integers with a subset relation is $(\mathbb{P}(\mathbb{Z}), \subseteq)$.

Standard means to infer loop invariant is to compute fixed points of function over this lattice structure [70, 164]. The fixed point of the function $F(X)$ gives the set X that satisfies $F(X) = X$. The loop invariant is given by $(x \geq 4 \wedge x \leq 10 \wedge x \equiv 0 \pmod 2)$. The set of values of x satisfying this loop invariant is the least fixed point.

Abstract interpretation of a program computes a fixed point of an abstract function (called abstract transformer), over an abstract lattice. Assuming an Interval lattice $Intv$ which maps a set of integer values of a variable to the smallest interval that contains it, the function $F(X)$ is abstracted over an Interval lattice as shown below.

$$F^\sharp([a,b]) = [4,4] \sqcup ([a,b] \sqcap [-\infty, 10]) +_{Intv} [2,2]$$

Control Location	Iteration 1	Iteration 2	Iteration 3	Iteration 4
$n1$	$x\colon \top$	$x\colon \top$	$x\colon \top$	$x\colon \top$
$n2$	$x\colon [2,2]$	$x\colon [2,2]$	$x\colon [2,2]$	$x\colon [2,2]$
$n3$	$x\colon [4,4]$	$x\colon [4,6]$	$x\colon [4,8]$	$x\colon [4,10]$
$n4$	$x\colon [6,6]$	$x\colon [6,8]$	$x\colon [6,10]$	$x\colon [6,10]$
$n5$	$x\colon \bot$	$x\colon \bot$	$x\colon \bot$	$x\colon [10,10]$
Error	$x\colon \bot$	$x\colon \bot$	$x\colon \bot$	$x\colon \bot$
Safe	$x\colon \bot$	$x\colon \bot$	$x\colon \bot$	$x\colon [10,10]$

Figure 2.7: Fixed point computation of program of Figure 2.6

The initial values of x is $[4,4]$ which is obtained by interval analysis of the static analysis equation, $S_{n3} = post_{x:=x*x}(S_{n2})$. The function $F^\sharp$ computes an interval at each iteration where the interval below 10 is incremented by 2 and $(+_{Intv})$ denotes an addition operation in the Interval lattice. Figure 2.7 gives the fixed point computation of the loop of the CFG in Figure 2.6, over a lattice of intervals. Each column denotes an iteration of the fixed point computation which associates an interval with each location in the program. The initial value of x is $\top$ at $n1$, while the locations $n2$, $n3$, $n4$, $n5$, *Error*, *Safe* are considered unreachable. Each iteration of the loop computes a bound on the variable x. The interval $x\colon [4,10]$ at the loop head $n3$ in the last iteration is the loop invariant. In practice, the total number of iterations may be too large to reach a fixed point. Hence, techniques like widening and narrowing are used to accelerate convergence [69, 72].

2.6 Abstract Conflict Driven Clause Learning (ACDCL)

Abstract Conflict Driven Clause Learning [86, 87, 89, 122] is a lattice-based generalization of the CDCL algorithm, proposed by Haller et al. for application of abstract interpretation to design satisfiability algorithms. Haller's work present an understanding of CDCL-based SAT solver in terms of lattice structures, transformers and fixed point iteration. An objective behind this unification is to achieve uniform theoretical and practical treatment of the SAT/SMT solving and static analysis using lattice theoretic techniques. The expectation is that this will contribute to the development of new class of solvers and program analyzer technologies that are easy to develop and have improved performance. The content below is not new and gives a background of the ACDCL framework.

SAT solvers works on Partial Assignments Domain

Given a CNF formula φ, a *partial assignment* π is a truth assignment to a subset of variables of φ, that is a partial function in $Prop \to \mathbb{B}$.

Example 2.6.1. Consider a CNF formula $\varphi = (\neg p \vee q) \wedge (p \vee \neg q) \wedge (p \vee r)$ and a partial assignment $\pi = \{p : f, q : t\}$. Clearly, φ is not satisfied since $(p \vee \neg q)$ evaluates to *false* even without assignment to variable r. The partial assignment cannot be extended further since every extension of π would evaluate to *false* due to unsatisfiability of the clause $(p \vee \neg q)$. This leads to a *conflict* state, denoted by $\bot$.

We model partial assignments as a total function, $\pi : Prop \to \{t, f, \top\}$ over partial assignments domain *PAsgn*, where for each variable p, $\pi(p) = \top$ if π is undefined on p. The *partial assignments domain* is given by, $PAsgn \triangleq (Prop \to \{t, f, \top\}) \cup \{\bot\}$. The set *PAsgn* contains a special element $\bot$ that denotes *conflict*. The ordering of elements in partial assignments domain, $(PAsgn, \sqsubseteq)$, is $t \sqsubseteq \top$ and $f \sqsubseteq \top$ with the empty set denoted by $\bot$.

Figure 2.8 presents a partial assignments domain over two variables, `p` and `q`. The element `(p:t)` in *PAsgn* denotes a partial assignment in which `p` is *true* and the remaining variables are mapped to $\top$. The Galois connection between the concrete domain of

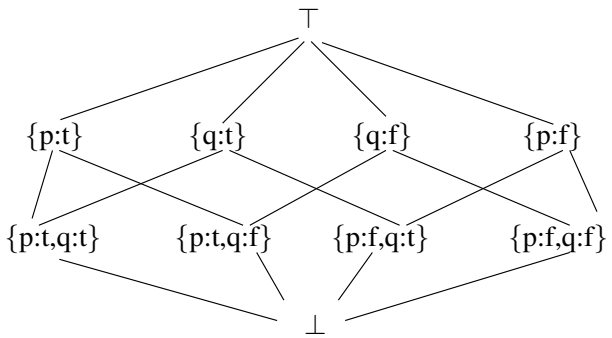


Figure 2.8: Partial Assignments Domain over two variables {`p`, `q`}

assignments *Asgn* and the overapproximating partial assignments domain, *PAsgn*, is given by

$$(\mathbb{P}(Asgn), \subseteq, \cup, \cap) \underset{\alpha_{PAsgn}}{\overset{\gamma_{PAsgn}}{\leftrightarrows}} (PAsgn, \sqsubseteq, \sqcup, \sqcap)$$

The abstraction function, α_{PAsgn}, and concretization function, γ_{PAsgn}, are defined by,

$$\alpha_{PAsgn}(\emptyset) \triangleq \bot, \quad \alpha_{PAsgn}(C) \triangleq \{p \mapsto \bigsqcup\{\tau(p) \mid \tau \in C\} \mid p \in Prop\}, \text{for } C \neq \emptyset$$
$$\gamma_{PAsgn}(\bot) \triangleq \emptyset, \quad \gamma_{PAsgn}(\pi) \triangleq \{\tau \in Asgn \mid \tau(p) \sqsubseteq \pi(p), \forall p \in Prop\}$$

Abstract Interpretation of Satisfiability

Haller et al. [86, 87] define fixed point characterizations of the models and countermodels of a formula and express satisfiability and validity in terms of these fixed points.

Let *Prop* be a set of propositional variables. A literal is a variable or its negation. Let $\mathbb{B} \triangleq \{t, f\}$ be the set of truth values. Let φ be a CNF formula. Let $Asgn \triangleq Prop \to \mathbb{B}$ be the set of concrete assignments. A concrete domain over *Asgn* is given by $(\mathbb{P}(Asgn), \subseteq, \cup, \cap)$. Properties of φ can be expressed with the following transformers.

A *concrete model transformer* (f^{φ}_{mod}) over a set of assignments Z removes all countermodels of φ from the set Z such that it contains only those assignments that satisfy φ. Dually, a *concrete countermodel transformer* (f^{φ}_{cmod}) over a set of assignments Z removes all models of φ from Z such that it contains only those assignments that does not satisfy φ. More formally,

$$f^{\varphi}_{mod}(Z) \triangleq \{\tau \in Z | \tau \models \varphi\}, \quad f^{\varphi}_{cmod}(Z) \triangleq \{\tau \in Z | \tau \not\models \varphi\}$$

We define a *universal countermodel transformer*, f^{φ}_{ucmod}, over a set of assignments Z, that adds all countermodels of φ to Z.

$$f^{\varphi}_{ucmod}(Z) \triangleq \{\tau \in Asgn | \tau \not\models \varphi \text{ or } \tau \in Z\}$$

A fixed point characterization of satisfiability can be expressed by suitably exploiting the algebraic properties of these transformers, f_{mod} and f_{cmod}. The concrete model and countermodel transformers have the following properties.

1. f^{φ}_{mod} and f^{φ}_{cmod} are idempotent.
2. $f^{\varphi}_{mod}(Asgn)$ is the set of models of φ and $f^{\varphi}_{cmod}(Asgn)$ is the set of countermodels of φ.
3. There exists Galois connection as shown below.

$$\mathbb{P}(Asgn) \underset{f^{\varphi}_{mod}}{\overset{f^{\varphi}_{cmod}}{\leftrightarrows}} \mathbb{P}(Asgn)$$

A formula φ is unsatisfiable iff $f^{\varphi}_{mod}(Asgn)$ is empty and $f^{\varphi}_{cmod}(Asgn)$ contains set of all assignments. Further, a formula φ is unsatisfiable iff the greatest fixed point of f^{φ}_{mod}, written $\mathsf{gfp}(f^{\varphi}_{mod})$, is empty.

Given a concrete domain of assignments, $(\mathbb{P}(Asgn), \subseteq, \cup, \cap)$, let $(O, \sqsubseteq, \sqcup, \sqcap)$ be an overapproximation of the concrete domain of assignments and $(U, \preceq, \curlyvee, \curlywedge)$ be an underapproximation of concrete domain of assignments. The subset ordering $\subseteq$ of assignments are refined such that $a \sqsubseteq b$ implies $\gamma(a) \subseteq \gamma(b)$ and $p \preceq q$ implies $\gamma(p) \subseteq \gamma(q)$. The Galois connection between the concrete domain of assignments and the approximating domains, O and U are given by

$$(\mathbb{P}(Asgn), \subseteq, \cup, \cap) \underset{\alpha_O}{\overset{\gamma_O}{\leftrightarrows}} (O, \sqsubseteq, \sqcup, \sqcap)$$

$$(\mathbb{P}(Asgn), \supseteq, \cup, \cap) \underset{\alpha_U}{\overset{\gamma_U}{\leftrightarrows}} (U, \succeq, \curlyvee, \curlywedge)$$

Let $f^{\varphi}_{aomod} : O \to O$ be a sound abstract overapproximation of f^{φ}_{mod} and $f^{\varphi}_{aucmod} : U \to U$ be a sound abstract underapproximation of concrete countermodel transformer f^{φ}_{cmod}. The abstract transformers satisfy the following constraints.

$$f^{\varphi}_{mod} \circ \gamma_O \subseteq \gamma_O \circ f^{\varphi}_{aomod}, \qquad f^{\varphi}_{cmod} \circ \gamma_U \supseteq \gamma_U \circ f^{\varphi}_{aucmod}$$

Now, following the fixed point transfer theorem in abstract interpretation [69], a formula φ is unsatisfiable iff $\gamma_O(gfp(f^{\varphi}_{aomod}))$ is empty.

2.6.1 Application of Lattice-theoretic Characterization of CDCL

SAT/SMT-based model checkers use bit-blasting for reasoning about the properties of a design, hence they are precise but inefficient. The inefficiency is mainly due to the capacity limitations of the underlying SAT/SMT solvers. On the other hand, static analysis using abstract interpretation typically analyze a design by computing a fixed point over non-distributive abstract domains, which soundly overapproximate the concrete fixed point. The use of non-distributive abstract domains make abstract interpreter efficient but imprecise. The imprecision is typically due to the overapproximate analysis resulting from the control-flow joins, loop widening or through the use of imprecise abstract transformers. In practice, some degree of imprecision in static analysis tools can be reduced by performing aggressive manual partitioning or introducing new abstract domains or through domain refinement [70, 106]. By contrast, CDCL-based SAT solver operates on non-distributive structures such as partial assignments, but CDCL solver can automatically recover from imprecision using decision and learning techniques. The lattice-theoretic characterization of CDCL (in Section 2.6) may be perceived as a way to automatically improve the precision of the fixed point analysis over non-distributive abstract domains. To this end, this dissertation shows that the precise abstract interpretation of the software netlist design of the RTL can be automatically performed using a CDCL-style analysis over an abstract lattice of program traces. The analysis combines the overapproximation of the greatest fixed point computation for finding counterexample with the underapproximation of the least fixed point computation for finding a generalized conflict reason from the partial safety proof. This combination of overapproximation and underapproximation of fixed point computations implicitly partition the traces of the software netlist in a way that enables precise reasoning of disjunctive properties over non-distributive abstractions of concrete traces. In a nutshell, the precise reasoning architecture of the CDCL solver is lifted over non-distributive numerical abstract domains using the framework of abstract interpretation, for formal property verification

of RTL designs. It is worth emphasizing that the proposed technique does not bit-blast the design unlike SAT/SMT-based verification tools, nor does it employ expensive abstract domains for the analysis.

Chapter 3

Verilog to C Translation

3.1 Hardware Design in Verilog

Verilog is a hardware description language (HDL) that can be used to describe the behavior of electronic hardware at varying levels of abstraction. Verilog can simulate models at various levels namely the *algorithm, RTL, gate* and *switch* level. Verilog also offers two different methods of circuit specification, which are behavioral and structural specifications. Structural specification consist only of primitives or module definitions and their instances and thus it allows designers to describe a digital system as a hierarchical interconnection of these modules/primitives. In contrast, behavioral Verilog can be used to describe designs at a high level of abstraction using arithmetic expressions, procedural assignments, or other Verilog control-flow structures. Although it is obvious that C and Verilog have many common features (for example, the `if-else` statement in Verilog is similar to C), especially in the syntax and the semantics of the operator definition, there are other aspects which differentiate these two design models. Notably, C supports use of pointers and recursion, which Verilog does not support. On the other hand, Verilog has bit selection operators to select specific bits from vectors and also concatenation operators to concatenate expressions to form vectors which C does not support.

However, the major difference lies in the fact that Verilog statements like the initial block, the always block, the generate statement, procedural assignment (blocking, non-blocking) and continuous assignment are not supported in C. It is also noteworthy to mention that the data model of Verilog supports 4-valued data-types which is significantly different from C. These non-trivial constructs, combined with parallelism, make the translation of Verilog to C challenging.

The notion of compiling from Verilog to C/C++ has been around for a long time. Many companies including Intel and Synopsys have tools dating back to the 90's that compile HDLs such as Verilog, VHDL or their proprietary HDL to C or C++ and then use GCC compiler for generating executable code. They use these C/C++ code for scalar simulation [219], i.e. non-symbolic simulation. The practice is well known today and can be seen in well known simulators such as VCS [1]. The main difference is that our work allows for symbolic verification instead of scalar simuation.

Verilog has two different semantic interpretations: *simulation semantics* [94, 178, 188] and *synthesis semantics* [112]. Formal property verification tools which work on register transfer level descriptions use synthesis semantics in order to synthesize a Verilog design into netlist representations. In this dissertation, we only use the synthesis semantics of Verilog.

Plan of the chapter

The rest of the chapter is organized as follows. Section 3.2 discusses the related work. Section 3.3 describes the Verilog to C translation in details. Section 3.4 explains the limitation of the V2C translator. The implementation and experimental evaluation is discussed in Section 3.5. Section 3.6 concludes the chapter.

Claim of Novelty

This chapter presents an automatic translation from Verilog at register transfer level into software program which is expressed in C language. Unlike previous approaches for generating C/C++ abstract models from Verilog RTL, the translation presented in this dissertation is unique in three ways – 1) The software model is bit-precise but not cycle-accurate, 2) The purpose of our translation is assertion-based verification and not simulation, 3) Formal RTL verification tools have never synthesized a software from RTL circuit for formal property checking. This dissertation presents the formal property verification of hardware RTL designs via translation to software.

3.2 Related Work

There have been several attempts at building compilers/converters that can be used to convert the Verilog models into another model. There exist many compilers that are used to convert Verilog to other hardware specification languages such as VHDL or SystemC,

[1] https://www.synopsys.com/verification/simulation/vcs.html

and vice versa. These converters are built mainly to address compatibility issues and for gaining simulation speed-ups. There are other motivations for converting Verilog models into other forms in order to use the target model as an intermediate representation that can later be used for equivalence checking, analysis and in other application specific domains.

The tool Verilog2SystemC [2] translates the input behavioral Verilog design into SystemC. Verilator [3] is a compiler that accepts synthesizable Verilog and translate it into an equivalent C++ program. The major drawback of Verilator is that it works with a very restrictive synthesizable subset of Verilog and generates excessively large C++ output code. The work of [116] develops a Verilog to C compiler, VTOC, preserving only the synthesis semantics of the input Verilog design. But the tool VTOC is not available publicly.

3.3 Verilog to C Translation

In this chapter, we show that the hardware circuit given in Verilog RTL can be translated into software which is expressed in C language. We will call the software model of RTL as *software netlist*. To this end, we present a Verilog to C translator tool which we name V2C. Given a Verilog RTL design, V2C uses the synthesis semantics of Verilog to automatically generate a software netlist in C language.

Software Netlist

A *software netlist SN* is a sequential program which is defined by four-tuples $\langle L, A, l_0, l_e \rangle$, where L is the finite set of control locations, $l_0 \in L$ is the initial location, $l_e \in L$ is the error location and $A \subseteq L \times M \times L$ is the control flow automaton. The notion of error location l_e in *SN* is relevant for the purposes of property checking. The edges in A are labelled with a quantifier-free first-order formula M over the variables of *SN*. The formula M is defined by five-tuples $\langle In, Out, Seq, Comb, Asgn \rangle$, where *In*, *Out*, *Seq*, *Comb* are *input*, *output*, *sequential* or *state-holding* and *combinational* or *stateless* variables, respectively. *Asgn* is a finite set of assignments to *Out*, *Seq* and *Comb*, which are formally defined below.

[2] http://www.edautils.com/verilog2systemc.html
[3] http://www.veripool.com/verilator.html

Here, bv_{var}, bv_{const} denotes bit-vector of variable width and constant width, respectively.

$$
\begin{array}{lll}
Asgn & ::= & (V := Bv) \mid (V := Bool), V \in \{Comb, Seq, Out\} \\
Bv & ::= & bv_{const} \mid bv_{var} \mid ITE(Bool, Bv, Bv) \mid \\
 & & bv_{op}(bv_1, \ldots, bv_n), bv_i \in Bv, \\
 & & bv_{op} \in \{n\text{-ary operators over bitvectors}\} \\
Bool & ::= & true \mid false \mid \neg Bool \mid \\
 & & Bool \wedge Bool \mid Bool \vee Bool \mid \\
 & & Bv\ relop\ Bv, relop \in \{<, \leq, =, >, \geq\}
\end{array}
$$

3.3.1 V2C – The Verilog RTL to C Translator

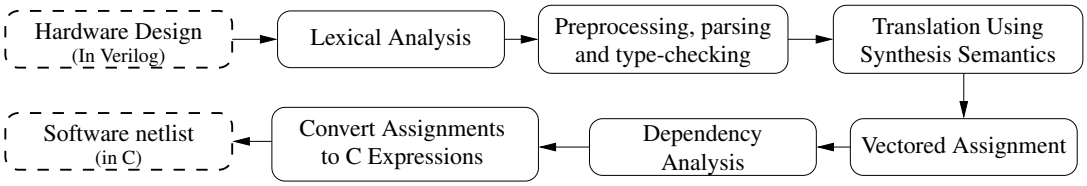


Figure 3.1: Translation stages in V2C

Figure 3.1 illustrates the translation steps of V2C. The front-end phase parses the input Verilog, performs macro-preprocessing and type-checking. The front-end supports the 1364-2005 IEEE Standard for Verilog [4]. The front-end generates a type-annotated parse-tree, which is passed to the translation phase. During this phase, V2C uses the synthesis semantics to translate Verilog into C, which is shown by the box labelled "Translation Using Synthesis Semantics" of Figure 3.1. Thus, the resultant software netlist in C mimics the structure of the synthesized hardware. The complex bit-manipulating operations in Verilog is translated into equivalent C expressions using combination of bit-wise C operators such as shift and mask operators, which is denoted by the box labelled "Vectored Assignment". The tool then performs the global dependency analysis to determine the inter-modular and intra-modular dependencies, which is shown by the box labelled "Dependency Analysis". The translation phase is followed by the code-generation phase, where the intermediate expressions and translated module items are converted into C expressions, shown by the box labelled "Convert Assignments to C Expressions". Note that we refrain from any optimizations or abstractions during the translation process to obtain a correct and trustworthy output.

3.3.2 Translation Rules

A point to note about the execution of Verilog statements is that they are concurrent in nature, which implies that there is no pre-defined execution order. By contrast, in C, the statements are executed in sequential manner. We now present syntactic translation rules for translating

synthesizable Verilog RTL into an equivalent software netlist, expressed in C language. We will use Figure 3.3 to desribe the translation. Figure 3.3 gives a D-Flipflop implementation in Verilog RTL (on the left). The right side of Figure 3.3 gives the equivalent software netlist design of the D-Flipflop.

Data Model

The data model in Verilog is significantly different from C. Each bit of a C integer value can have only two states, namely 0 and 1. Bits in Verilog HDL can take one of four values, namely 0, 1, *X* and *Z*. A value of 0 represents low voltage and value of 1 represents high voltage. Further, the values *X* and *Z* represent an unknown logic state and a high impedance value, respectively. For example, for `inout` signals, the value *Z* can be synthesized. In this case, the `inout` signal are typically synthesized to a tristate buffer. However, if all of the `inout` pins are connected to a single wire, then the synthesizer simply infers a multiplexer logic.

Translation Rule: The simplest synthesis semantics for X is treating it as a don't-care assignment, which allows the synthesis tool to choose a 0 or 1 to further improve logic minimization. The X values are translated to non-deterministic assignment in the software netlist. However, *Z* value has no corresponding state in C language. So, when a *Z* value is encountered, the translator flags a warning message and stops the translation.

Module and Module Instantiations

The general syntax of module declaration is shown below:

```
module <module_name>(<input_list>, <output_list>);
  input <input_list>;
  output <output_list>;
endmodule
```

The general syntax of module instantiation is as follows:

```
<module_name><instance_name>(<port_list>);
```

An alternative method for module instantiation which does not depend on the port ordering list is shown below:

```
<module_name><instance_name>(.<port_name>(ioname), ...);
```

The communication between modules takes place through ports, which are signals listed in the parameter list at the top of the module. Ports can be of type in, out, and inout. The outputs of the behavioral block are mainly of reg data type because if wire was used, it would prevent the outputs of the behavioral block from being used by any other communicating

block. The following code illustrates the module hierarchy in Verilog and inter-module communication:

```
module M1(...);
...
endmodule

module M2(...);
 M1 m1(...);    // Instantiates M1
endmodule

module M3(...);
 M1 m1(...); instantiates M1
 M2 m2(...); Instantiates M2
endmodule
```

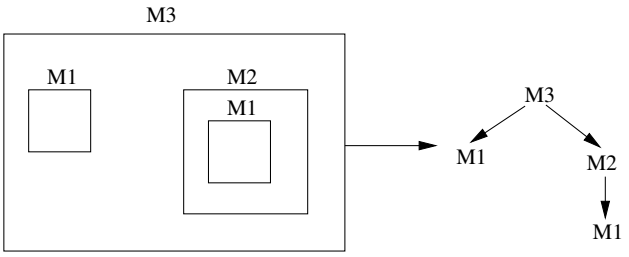


Figure 3.2: Modular hierarchy in Verilog

Figure 3.2 demonstrates the Verilog module hierarchy and the interaction between different modules. Here, the module *M1* instantiated inside top-level module *M3* and *M1* instantiated inside *M2* are two different entities. The module behaviors are exactly the same in both cases, but differ in the date they operate on.

Translation Rule: The module in Verilog is translated to the function in software netlist. The module hierarchy in Verilog is also preserved in the software netlist. The parameters of a module are translated to function arguments. Figure 3.3 gives an example of module hierarchy for D-Flipflop where the top-level module `top` instantiates module `ff`. The software netlist (on the right) preserves the module hierarchy of the Verilog RTL in Figure 3.3. Additionally, the software netlist implements a `main` routine that invokes the initial blocks `initial_ff, initial_top`, constructs a `while(1)` wrapper that calls `top` to model sequential behavior and contains the assertions. The modeling of System Verilog Assertions in software netlist is described in Section 4.5.

Verilog

```
module top(clk,Din,En,Dout);
 input clk,Din,En;
 output Dout;
 wire cs; reg ns;

 initial begin
   ns = 0;
 end

 assign Dout = cs;
 always @(Din or cs or En) begin
 begin
  if (En) ns = Din;
  else ns = cs;
 end
 end
 ff ff(clk,ns,cs);

 assert property (Din==1&&En==1 |-> ##1 Dout==1);
 assert property (Din==0&&En==1 |-> ##1 Dout==0);
endmodule

module ff(clk,Din,Dout);
 input clk, Din;
 output Dout;
 reg q;
 initial begin
   q = 0;
 end
 assign Dout = q;
 always @(posedge clk)
   q <= Din;
endmodule
```

Software netlist

```
_Bool nondet_Bool();
struct state_ff{
  _Bool q;
};
struct state_top{
 _Bool ns;
 struct state_ff sff;
} stop;

_Bool ff(_Bool clk,
_Bool Din, _Bool *Dout){
  stop.sff.q = Din;
  *Dout = stop.sff.q;
  return;
}

void top(_Bool clk, _Bool Din,
_Bool En, _Bool *Dout)
{
 _Bool cs;
 if(En)
  stop.ns = Din;
 else
  stop.ns = cs;
 ff(clk, stop.ns, &cs);
 *Dout = cs;
 return;
}

void initial_top() {
  stop.ns = 0;
}

void initial_ff() {
  stop.sff.q = 0;
}

int main() {
 _Bool clk,En,Din,out;
 initial_top();
 initial_ff();
 while(1) {
  Din = nondet_Bool();
  En = nondet_Bool();
  if(En==1 && Din==1) {
    top(clk,Din,En,&out);
    assert(out==1);
  }
  if(En==1 && Din==0) {
    top(clk,Din,En,&out);
    assert(out==0);
  }
 }
 return;
}
```

Figure 3.3: Verilog RTL and software netlist

Registers and Wires

Structural data types, also called *nets*, are used to model hardware connections between circuit components. The two most common structural data types are wire and reg. The wire nets are used for physical connections and act like real wires in circuits. They do not store a value and must be driven by a driver. The initial value of wire is Z, if it is not driven externally. Unlike a C variable, which stores its present value until it is explicitly assigned a new value, the value of wire variables changes continuously as the input value changes. So, the value of Verilog wire has to be evaluated continuously. On the other hand, the state-holding elements, denoted by *reg* type, retain their values until another value is assigned to them and appears like a register hardware component. The initial state of a reg is X (unknown). The general syntax of register and wire declaration are as follows:

```
reg [msb:lsb] reg_variable_list;
wire [msb:lsb] wire_variable_list;
```

Translation Rule: The state-holding elements of a module are declared in a structure data-type in the software netlist. This structure can be nested, that is, if a module instantiates another module, and both contains register type elements, then the top-level structure contains the state-holding elements of the top-level module and the inner structure contains the state-holding elements of the module that is instantiated inside the top-level module.

Figure 3.3 demonstrate an example of reg data type and the corresponding structure data-type in the software netlist. Here, the structure corresponding the module `top` is `state_top` and the structure corresponding to the module `ff` is `state_ff`. Since, the module `top` instantiates module `ff`, so the structure `state_top` contains structure `state_ff`.

The wire in Verilog are declared as automatic variables in software netlist. For example, `wire cs;` of module `top` in Verilog RTL is translated to local variable `_Bool cs` of function `top` in the software netlist. Any assignment to the wire is handled in the same way as the continuous assignment statements in Verilog, which is explained in Subsection 3.3.2.

The declaration of Verilog data types (like register and wire) may vary, depending on data width. The following declaration in Table 3.1 gives the data-widths in Verilog RTL and the equivalent or nearest data type in the software netlist for linux systems. In this dissertation, we do not handle bit-width longer than 128 bits.

Parameters

Parameters are constants which are typically used to specify the width of variables. The general syntax of Parameter is shown below:

Verilog Bit Width	C type
0–8	char
8–16	short
16–32	int
32–64	long int
64–128	unsigned __int128

Table 3.1: Data-width handling for Linux x86 systems

```
parameter identifier = constant_expression;
For example, parameter size = 8;
```

Translation Rule: Table 3.2 gives the translation rule for Parameters.

Verilog	Software netlist
parameter	const int

Table 3.2: Parameter

Translation Rule: Table 3.3 gives the translation rule for Constants.

Verilog	Software netlist
‘define	#define

Table 3.3: Constant

Input, Output, and Inout

The input port and inout port are used to describe the input and bidirectional ports of a module and are of type wire, but an output port can be configured as wire or reg. The default type of output port is wire. The general syntax of port declaration is shown below:

```
input [msb:lsb] input_port_list;
output [msb:lsb] output_port_list;
inout [msb:lsb] inout_port_list;
```

Translation Rule: The input ports of a module are passed by value to the corresponding function in the software netlist while the output ports and inout ports are passed by reference using pointer variables. If the output port is also a register declared as `output reg`, then it is also declared in the structure data-type for that corresponding module.

Figure 3.3 gives the handling of the input and output ports. The output port `Dout` of module `top` is translated to a pointer variable which is passed by reference in the software netlist. The input ports `clk, Din, En` are passed by value.

Always block

Always blocks are the concurrent statements which execute continuously during the simulation process and all always blocks that are present inside a module are executed simultaneously. The statements enclosed inside the always block within begin ... end construct are executed sequentially. Each always block has a sensitivity list and when a variable in the sensitivity list changes, the always block wakes up and executes its enclosed statements. The general syntax of always blocks is shown below.

```
always @(event_1 or event_2 or ...)
begin
... statements ...
end
```

For example,

```
always @(a or b)//level-triggered if a or b changes levels
always @(posedge clk)//edge-triggered on positive clk edge
```

Translation Rule: Figure 3.4 shows the modeling of the rising and falling edge of a clock in software netlist. The call to RTL module *M* in the positive edge of clock is modeled using the software variables *clk* and clk_{old} which flips its value at every iteration of the *while* loop. Thus, when $clk = 1$ (positive clock edge), then the call to function *M* is invoked in the software netlist.

Similar strategy could be implemented for level triggered always blocks, such as $always@(x)$, which executes its enclosing statements only when the signal x changes levels. This can be modeled in software netlist using an *if* statement such as $if(x_{old} \neq x)$, where x_{old} contains an old value of x and the current updated value is x.

The cycle-accurate model of the software netlist, shown in Figure 3.4, explicitly models the behavior of the clock, which may be essential for simulation purposes. However, the purpose of our translation is to perform formal verification, which requires a model of the synthesized hardware. Thus, a cycle-accurate model may not be necessary for this purpose.

Hence, an edge-triggered or level-triggered always block is translated to a sequential code block in the software netlist. The procedural assignment statements inside the always block are retained and translated to equivalent statements in the software netlist. Figure 3.3 gives the translation of a level-triggered always block in module `top` and an edge-triggered always block in module `ff`.

Verilog RTL	Software netlist
`module M(in, out);` `input in;` `output out;` `always@(posedge clk)` `begin` `  out <= in;` `end` `endmodule`	`int main()` `{` `  bool in, out;` `  bool clk, clk_old;` `  while(1) {` `  clk_old = clk;` `  clk = !clk;` `  //definition of posedge clk` `  if (clk_old==0 && clk==1)` `  M(in,&out);` `  }` `}` `void M(bool in, bool *out)` `{` `  *out = in;` `}`

Figure 3.4: Verilog RTL on the left and software netlist with explicit modeling of clock on the right

Initial blocks

The behavior of an initial block is same as the always blocks, except that they are executed exactly once, before the execution of any always block. The order in which the initial blocks are executed is uncertain and cannot be determined. The general syntax of the initial statement is shown below.

```
initial begin
... statements ...
end
```

Translation Rule: The statements inside the initial block will be executed only once, so the initial block is modeled by a function with prefix `initial` which is called once from the `main` routine. The non-deterministic execution order of the initial blocks are translated by invoking the initial blocks inside the main routine following the module hierarchy, that is, the initial blocks of the top-level module is called first followed by the initial blocks of the sub-modules that are instantiated inside the top-level block and so on. For example, Figure 3.3 gives the translation of the initial blocks in Verilog RTL following the module hierarchy to the corresponding functions `initial_top` and `initial_ff` in the software netlist.

Procedural Assignment

Procedural assignments are used within Verilog procedures like always and initial blocks. The left-hand side of a procedural assignment statement can be of type reg or integer variables, while the right-hand side of the assignment is an expression. Procedural assignments are of two types: *blocking* and *non-blocking*.

Non-blocking assignment	Blocking assignment	Continuous assignment
reg [7:0] x,y,z; **wire** in = 1'b1; **always** @(**posedge** clk) **begin** x <= in; y <= x; z <= y; **end**	**reg** [7:0] x,y,z; **wire** in = 1'b1; **always** @(**posedge** clk) **begin** x = in; y = x; z = y; **end**	**wire** in; **reg** a,b,t; **wire** a = in; **wire** c = b; **wire** d = c; **always** @(**posedge** clk) **begin** b <= a; t <= b; **end**
struct smain { **unsigned char** x,y,z; } sm; **unsigned char** xs,ys,zs; bool in = 1; *// save register variables* xs=sm.x; ys=sm.y; zs=sm.z; *// update register variables* sm.x = in; sm.y = xs; sm.z = ys;	**struct** smain { **unsigned char** x,y,z;}sm; bool in = 1; *// clocked block* sm.x = in; sm.y = sm.x; sm.z = sm.y;	**struct** smain { bool a,b,t; } sm; bool in,c,d,as,bs,cs,ds,ts; sm.a = in;*//continuous assign* *// save register variables* as=sm.a; bs=sm.b; ts=sm.t; *// clocked block* sm.b = as; sm.t = bs; *// continuous assignment* c = sm.b; d = c;

Figure 3.5: Tanslation of non-blocking, blocking and continuous assignments

- Blocking Assignment: Blocking statements are executed in sequential order and so blocking statements are used for combinational circuits.
- Non-Blocking Assignment: The order of non-blocking statements does not matter. Thus, Non-blocking statements are used for sequential circuits.

The syntax of procedural assignment statement is shown below:

```
variable = expression  --- Blocking Statement
variable <= expression --- Non-blocking Statement
```

Translation Rule: Blocking assignments are executed in sequential order. Thus, the translation of blocking assignments is straight-forward. The same execution order of the blocking assignments are retained in the software netlist. The effect of blocking assignment is visible immediately, whereas the effect of non-blocking assignments is delayed until all events triggered are processed. The parallelism from non-blocking assignment statements are modeled using *shadow variable update technique*. This technique stores the value of all register variables into auxiliary variables in the beginning of the function. Each read access to the register variables are then replaced by these auxiliary variables. This ensures that a non-blocking assignment to a register do not influence subsequent read of that register. Figure 3.5 illustrates the translation of procedural assignments (given at the top) to the equivalent software netlist (given at the bottom). Note that the code-snippets are partial and are only used for illustrating the translation of procedural assignments.

Continuous Assignment

The continuous assignment is used to assign a value onto a wire in a module. Continuous assignment statements are concurrent statements which are executed continuously during simulation. It is done outside of always or initial blocks. It uses an explicit assign statement or assigns a value to a wire during its declaration. The ordering of assign statements does not matter and thus any change in the right-hand-side inputs will immediately change the left-hand-side output. The general syntax of assign statements is shown below.

```
wire wire_variable = value;
assign wire_variable = expression;
```

Translation Rule: The ordering of the continuous assignment statement is determined by the dependency analysis (see Section 3.3.3) which identifies the nets that are assigned by an expression or value and occur in the right-hand side of other continuous statements. Thus, the ordering of the continuous statements in the software netlist follow the *Read-After-Write* ordering. Figure 3.3 gives the translation of the continuous assignment statement `Dout = cs` in the module `top`. Note that, the wire `Dout` reads from wire `cs` which is in turn updated by the module `ff`. Hence, following the Read-After-Write ordering, the continuous assignment `Dout = cs` is placed after the call to the function `ff` in the software netlist. Figure 3.5 also gives several examples of the translation of continuous assignment statement in Verilog.

3.3.3 Dependency Analysis

A hardware circuit specified in Verilog RTL may have two types of dependencies – 1) *Intra-modular dependency* and b) *Inter-modular dependency*.

3.3.3.1 Intra-Modular Dependency Analysis

The intra-modular dependencies may occur due to the communication between combinational blocks (continuous assignments) and sequential or clocked procedural blocks. We illustrate three different scenarios that summarize the various sources of intra-modular dependencies in Verilog and show how these dependencies are handled in a software netlist. Other forms of intra-modular dependencies are simply variants of these three cases and can be handled appropriately. Figure 3.6– 3.8 graphically illustrate the intra-modular dependencies. Here, box denote an input, a bold circle denote a wire and a normal circle denote a latch. The bold edges denote the dependencies between two wires or a latch and a wire with the arrow pointing towards the wire and normal edges denote the dependencies between two latches or a wire and a latch with the arrow pointing towards the latch. The dotted

Verilog	Dataflow Graph	Software netlist
`module main();` `wire x;` `wire [1:0] y;` `assign x=1'b1;` `assign y=x+1'b1;`		`int main() {` `bool x;` `unsigned char y;` `x=1;` `y=(x+1)&0x3;` `}`

Figure 3.6: Dependencies between combinational elements

arrows denote the dependencies with an input. We describe the various scenarios and the corresponding translations below. Note that the code-snippets in Figure 3.6– 3.8 are partial and are only used for illustrating the handling of intra-modular dependencies.

Scenario A A wire, say x, assigned in a continuous assignment statement, say A, appears in the right-hand side of another continuous assignment statement, say B. This is illustrated in Figure 3.6.

Translation A The variable assignment A is placed before the other assignment B which reads x.

Scenario B A wire, say x, appearing in the right-hand side of a continuous assignment, say A, is driven by an always block. This is illustrated in Figure 3.7.

Translation B This gives an ordering where the continuous assignment is placed after the always block to capture the updated value of x.

Scenario C A latch, say x, appearing in procedural block, say A, is assigned directly by the input signal and x is then read inside another procedural block. This is illustrated in Figure 3.8.

Translation C The assignment to x is placed before the second procedural block that reads x.

3.3.3.2 Inter-Modular Dependency Analysis

Modules in Verilog communicate with each other through their input or output ports. Most practical designs are modular in nature, where the top-level module delegates specific tasks to the sub-modules. Modules and sub-modules executes *in-tandem*, that is, a module is *not* blocked when it invokes a sub-module.

An example of inter-modular communication is illustrated in Figure 3.9. The top-level module `main` invokes the sub-module `M`. The equivalent translation to software netlist

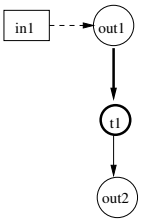
Verilog	Dataflow Graph	Software netlist
module M(in1, in2, out1, out2); **input** in1, in2; **output reg** out1, **output reg** out2; **wire** t1; **assign** t1=out1; **always** @(in1) **begin** out1 <= in1; **end** always@(t1) **begin** out2 <= t1; **end** **endmodule**		**int** M(bool in1, bool in2, bool *out1, bool *out2) { bool t1; *out1=in1; t1=*out1; *out2=t1; }

Figure 3.7: Dependencies between latches and combinational logic

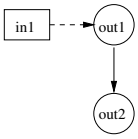
Verilog	Dataflow Graph	Software netlist
module M(in1, in2, out1, out2); **input** in1, in2; **output reg** out1, out2; **always** @(in1) **begin** out1 <= in1; **end** always@(out1) **begin** out2 <= out1; **end** **endmodule**		**int** M(bool in1, bool in2, bool *out1, *out2) { *out1=in1; *out2=*out1; }

Figure 3.8: Dependencies between latches

(shown on the right of Figure 3.9) preserves the module hierarchy of the Verilog RTL. An alternative scenario for inter-modular communication is the *combinational feedback loop* where the participating modules exchange combinational data until a stability condition is reached. This is demonstrated in Figure 3.12. However, for designs that exhibit *circular module dependency*, the module hierarchy may not be preserved during the translation. V2C cannot automatically translate designs with circular module dependency.

Verilog	Software netlist
`module M(clk, in,` `  out1, out2, out3);` `input clk,in;` `output reg out1,` `  out2, out3;` `wire t1;` `initial begin` `out1=0;` `out2=0;out3=0;` `end` `assign t1=out1;` `always @(in) begin` `out1 <= in;` `end` `always @(t1) begin` `out2 <= t1;` `end` `always @(posedge clk)` `begin` `out3 <= out2;` `end` `endmodule` `module main(clk);` `input clk;` `wire in,out1,` `  out2,out3;` `M m1(clk,in,out1,out2,out3);` `assert property` `  (in |-> ##1 out3);` `endmodule`	`struct state_M {` `  bool out1, out2, out3;` `};` `struct state_M sM;` `void initial() {` `  sM.out1=0; sM.out2=0;` `  sM.out3=0;` `}` `void M(bool clk, bool in,` `  bool *out1, bool *out2,` `  bool *out3) {` `  bool t1;` `  sM.out1=in;` `  t1=sM.out1;` `  sM.out2=t1;` `  sM.out3=sM.out2;` `  // update output` `  *out1=sM.out1;` `  *out2=sM.out3;` `  *out3=sM.out3;` `}` `int main()` `{` `  bool clk,in,` `  out1,out2,out3;` `  initial();` `  while(1) {` `  if(in) {` `  M(clk,in,` `  &out1,&out2,&out3);` `  assert(sM.out3);` `  }` `  }` `}`

Figure 3.9: Inter-modular dependency analysis

Bit-precise code generation:

V2C generates a bit-precise software netlist in C. The tool automatically handles complex bit-manipulating operators in Verilog like bit-select or part-select operators from a vector, concatenation operators, reduction OR and other operators. These operators are translated using a combination of bit-wise operators in C.

Figure 3.10 gives the Verilog RTL snippet (at the top) and the generated C expressions (at the bottom), which are combinations of bit-wise and arithmetic operators like bit-wise OR, AND, multiplication, subtraction, shifts and other C operators. Note that the code-snippets are partial and are only used for illustrating the translation of bit-wise operators in Verilog.

Bit-select	Part-select (SystemVerilog)	Concatenation
`wire [7:0] in1,in2;` `reg [7:0] out1,out2;` `out1[7:5] = in1[4:2];` `out2[6] = in2[4];`	`reg [31:0] in, out;` `for(i=0;i<=3;i++) begin` `out[8*i +: 8]=in[8*i +: 8];` `end`	`wire [7:0] in1, in2;` `reg [9:0] out;` `out = {in2[5:2],in1[6:1]};`
`unsigned char in1,in2;` `struct smain {` `unsigned char out1,out2; } sm;` `sm.out1 = sm.out1 & 0x1f |` `(((in1 & 0x1c)>>2)<<5);` `sm.out2 = (sm.out2 & 0xbf)|` `(((in2 & 0x10)>>4)<<6);`	`struct smain {` `unsigned int in,out; } sm;` `for(i=0;i<=3;i++) {` `x=8*i+(8-1); y=8*i;` `sm.out=(sm.out&!(2^31-2^y))` `|(sm.in&(2^31-2^y)); }`	`unsigned char in1,in2;` `struct smain {` `unsigned char out; } sm;` `sm.out = (((in2 >> 2)` `& 0xF) << 6)|` `((in1 >> 1) & 0x3F);`

Figure 3.10: Handling Bit-select, part-select from vectors and concatenation operator

Non-Synthesizable Constructs Not Supported

The following non-synthesizable constructs are not supported. These includes *delay, repeat, wait, fork ... join, event, deassign, force, release and time* constructs. The V2C translator simply ignores all non-synthesizable constructs.

Pseudo-code structure of Software netlist

The pseudo-code of the software netlist design is shown in Figure 3.11. V2C generate software netlist in C which preserves the module hierarchy of the Verilog RTL. Additionally, a `main` routine is generated that contains calls to the initialization blocks, a wrapper to invoke the top-level function of the software netlist which corresponds to the top-level module in the Verilog design and the assertions that are used for verification purposes. In Figure 3.11, we assume that the top level function is named `top` and it contains input (passed by value), and output (passed by reference). A global structure `state_elements_top` is declared that contains all state-holding elements of the function `top`. The body of this function gives the various elements such as continuous assignments, procedural blocks and calls to other functions (module instantiation) in the software netlist design.

3.4 Limitations of V2C Translator

V2C does not support automatic translation of RTL designs that contain the following design features.

Structure of software netlist

```
// parameter definition
// macro definition
struct state_elements_top
  // declare all state-holding elements
  // of the current module
};
struct state_elements_top stop;
int initial_block() { //initialization of nets }
// Input are passed by value and output by reference
int top (data_type input, data_type *output)
{
  // shadow variable declaration
  declare shadow variables for
  non-blocking assignments to
  the register elements

  // continuous assignments
  Place all continuous statements which
  are only dependant on input

  // always block
  Place the always block following the
  intra-modular dependencies
  // procedural statements are bit-precise

  // continuous assignments
  Place all continuous statements that
  are updated by the signals driven
  by the always block

  // Module instantiations
  Place all module instances
  with proper mapping of
  input and output ports

} // end of top module

int main() {
  // local varaibles
  declare all local variables
  which are passed to the design
  initial_block(); // call to initial block
  // check if the design is sequential.
  // if so, then put a  while(1) wrapper
  while(1) {
   // non-deterministic assignments
   assign non-deterministic values to inputs
   // call the design
   top(input, &output);
  }
  // Assertions
  Place the assertions here
} //end main
```

Figure 3.11: Structure of the Software netlist

1. Multi-clock designs
2. Transparent latches
3. Combinational feedback loop
4. Circular module dependency

The automatic translation of Verilog designs with multiple clock [96, 208], transparent latches [47], combinational feedback loop and circular module dependency are not handled by V2C. In this dissertation, we present an automatic translation and verification of the RTL designs that do not contain the above design features. We give an example of one such design feature. Figure 3.12 gives a Verilog design with combinational feedback loop on the left. The right side of Figure 3.12 gives the equivalent software netlist, which is translated manually. The Verilog module *foo* is partitioned into two separate modules during translation – a combinational module, *foo_comb* and a sequential module, *foo_seq*. The combinational exchanges between the Verilog modules *top* and *foo* continues until the values of a and z becomes stable, which in this case is equal to the width of these registers (32-bit). This stability condition is modeled in the software netlist design by invoking the procedure *foo_comb* from the *main* routine 32 times, followed by a call to the sequential module *foo_seq* which simply updates the register *y*. Determining this kind of stability condition completely automatically is difficult for large circuits. Note that the assertion given by $(y == loopback)$ in Figure 3.12 holds true in the RTL as well as the software netlist design, which is established using off-the-shelf hardware and software model checkers respectively.

Verilog	Software netlist

```
module M(c,a,z,y);
input c;
input [31:0] a;
output [31:0] z;
output reg [31:0] y;
assign z[0]=c;
assign z[31:1]=a[30:0];
always @(posedge clk)
y<=z;
endmodule

module top();
wire [31:0] loopback;
wire [31:0] y;
foo f(.c(1),.a(loopback),
      .z(loopback),.y());
assert property (y==loopback);
endmodule
```

```
struct state_elements_M {
  unsigned int y;
};
struct state_elements_M sM;

void foo_comb(bool c, unsigned int a,
unsigned int *z, unsigned int *y) {
  *z=(*z&0xfffffffe) | (c & 0x1);
  *z=(*z&0x00000001) | ((a&0x7fffffff)<<1);
}
void foo_seq(bool c, unsigned int a,
unsigned int *z, unsigned int *y) {
 sM.y = *z;
}
void top() {
  unsigned int y, loopback;
  for(i=1;i<33;i++)
    foo_comb(1,loopback,&loopback,&y);
  foo_seq(1,loopback,&loopback,&y);
  assert(y==loopback);
}
```

Figure 3.12: Handling combinational feedback loop

Verilog	Bit-level Netlist	Word-level Netlist	Software netlist
module top(Din,En, clk,Dout); **wire** cs; **reg** ns; **input** clk,Din,En; **output** Dout; **assign** Dout = cs; **always** @(Din **or** cs **or** En) **begin** **if** (En) ns = Din; **else** ns = cs; **end** ff ff(ns,clk,cs); **endmodule** **module** ff(Din,clk, Dout); **input** clk, Din; **output** Dout; **reg** q; **assign** Dout = q; **always** @(**posedge** clk) q <= Din; **endmodule**	***Variable Map:*** ***Inputs:***top.clk=0, top.Din=1, top.En=2, top.ff.CLK=3, top.ff.Din=4, **input**[0]=6 **input**[1]=7 **input**[2]=11 **input**[3]=12 ***Wires:*** top.Dout=11, top.ff.Dout=5, top.cs=12, top.ns=!10 ***Latch:*** top.ff.q=5 ***Transition constraints:*** !(var(5)&!var(12)) &!(!var(5)&var(12)) !(var(4)&!(var(2) &var(1))&!(!var(2) &var(7)))&!(!var(4) &!(!(var(2)&var(1)) &!(!var(2)&var(7)))) !(var(3)&!var(0))& !(!var(3)&var(0)) ***Next state functions:*** NEXT(top.ff.q)=var(4)	***State constraints:*** top.Dout==top.cs top.ff.Dout== top.ff.q top.ff.Din== top.ns top.ff.clk== top.clk top.ff.Dout== top.cs top.ns==top.En ? top.Din:top.cs ***Transisition constraints:*** next(top.ff.q)== top.ff.Din	_Bool nondet_Bool(); **struct** state_ff{ _Bool q;}; **struct** state_top{ _Bool ns; **struct** state_ff sff; } stop; _Bool ff(_Bool clk, _Bool Din, _Bool *Dout){ stop.sff.q = Din; *Dout = stop.sff.q; **return**; } **void** top(_Bool clk, _Bool Din, _Bool En, _Bool *Dout) { _Bool cs; **if**(En) stop.ns = Din; **else** stop.ns = cs; ff(clk, stop.ns, &cs); *Dout = cs; **return**;} **void** initial_top() { stop.ns = 0;} **void** initial_ff() { stop.sff.q = 0;} **int** main() { _Bool clk,En,Din,out; initial_top(); initial_ff(); **while**(1) { Din = nondet_Bool(); En = nondet_Bool(); top(clk,Din,En,&out); } **return**; }

Figure 3.13: Verilog RTL design and its bit-level netlist, word-level netlist and software netlist

Netlist at different levels of abstractions

Figure 3.13 gives the Verilog RTL design for D-Flipflop in Column 1. Column [2-4] gives the corresponding bit-level netlist, word-level netlist and software netlist. The bit-level netlist description is often stored in AIGER format, and the word-level description is stored in format that resembles SMT-LIB2 [4] format.

[4]http://smtlib.cs.uiowa.edu/language.shtml

3.5 Implementation and Experimental Evaluation

We have implemented V2C in C++ on top of the *CPROVER* framework [5]. V2C uses the Verilog parser from the *CPROVER* framework.

3.5.1 Benchmark and Tool Distribution

The source code of the tool V2C is available in the website `http://www.cprover.org/hardware/v2c/`. We also distribute a collection of benchmarks in Verilog RTL and C in the above website. These benchmarks can be used for simulation, property verification or equivalence checking. For property verification, the designs are annotated with safety properties. For equivalence checking, a miter is provided that passes symbolic inputs to the reference model and the implementation and checks their output using an assertion.

Command Line Options

Below is the command line option for running V2C. Note that the input file can be in either of the two formats, .v or .sv.

```
v2c <Verilog-file-name> --module <top-module-name> <output-C-file>
```

For example, assuming that the input file name of the design is `main.v` and the name of the top level module is `main`, the command for translation is given as follows.

```
v2c main.v --module main main.c
```

Table 3.4 gives the design statistics of hardware circuits in Verilog RTL and the equivalent software netlist in C. The benchmarks are classified into data-path intensive and control-path intensive designs. Column 1 reports the name of the circuit and Column 2-5 reports the number of Latches/Flip-flops, input ports, output ports and gate count for each circuit. Column 6 in Table 3.4 gives the number of lines of code (LOC) of the software netlist. We do not report the translation times since these times are negligible.

Regression Testing

We apply regression testing to check V2C tool for enhancements, error corrections and optimization features. Our regression test suite contain several RTL designs in SystemVerilog language. We classify our regression test suite into four separate categories – 1) Bit-precise category, 2) SystemVerilog language category, 3) Dependency testing category, and 4) Property checking category. All the designs in our regression test suite contain assertions

[5] http://www.cprover.org/

Circuit	Latches(L)/ FF	Input Ports	Output Ports	GATE Count	Software netlist (LOC)
Data-Path Intensive Designs					
UP-COUNTER	8 (FF)	3	1	66	22
FSM	4 (FF)	3	2	167	110
FIR	33 (FF)	3	1	15	198
SERIAL_ADDER	9(FF)	3	1	53	106
PIPELINED_ADDER	6(FF)	3	1	34	97
DIGITAL_AUDIO_VIDEO	10(FF)	10	8	269	329
BUFFER_ALLOCATION	5(FF)	5	2	100	102
Control-Path Intensive Designs					
ADPCM	196 (L)	5	4	3961	215
GCD CONTROLLER	4 (FF)	3	2	647	402
USB_PHY_IP	196 (FF)	8	10	977	862
CACHE COHERENCE	22 (FF)	9	6	112	828
ETHERNET_MAC_CONTROLLER	307(FF)	23	18	2818	9102
RCU	13(FF)	2	1	430	227
FIFO_CONTROLLER	37(FF)	5	1	83	140

Table 3.4: Design statistics for hardware (in Verilog RTL) and the software netlist (in C)

that are used for checking the corresponding design features or design functionality using off-the-shelf property checkers. The bit-precise category contains RTL designs with complex bit-manipulating operations such as concatenation, part-select, bit-extraction operator among others. The second category contains RTL benchmarks with various SystemVerilog language features that are used to stress test the front-end SystemVerilog parser of V2C. The third category contains Verilog RTL designs that exhibit complex intra-modular and inter-modular dependencies. The correctness of the dependency analysis framework in V2C is tested using designs of this category. The fourth category contains RTL benchmarks which are primarily used for property checking. These benchmarks contain block-level circuits such as USB controller, Ethernet controller, Buffer Allocation protocol, traffic-light controller among others. Chapter 4 demonstrates the various techniques for formal property checking of the RTL designs via translation into software netlist. Note that we consider the equivalence between the RTL and the software netlist that preserves 1) the input-output behavior, and 2) the outcome of all assertions must match in the RTL as well as in the software netlist design.

3.5.2 Application of V2C

We now demonstrate an application of V2C for formal property verification using two different case studies derived from real world Verilog circuits: *Traffic light controller* and *Buffer allocation protocol*.

Case Study 1: Verification of Traffic light controller

Figure 3.14 gives an example of Verilog RTL (on the left) and the equivalent software netlist in C (on the right). The RTL design has three properties. These properties are modeled in the software netlist using C assertions. The traffic light controller is an unsafe design. The bug is manifested exactly after 64 clock cycles in the Verilog RTL design due to the failure of the assertion (`Light_Sign` $\neq 2'd2$). In the software netlist, the bug is also detected after 64 unwindings of the `while(1)` loop in the `main` procedure. The other two properties are globally true. The Verilog RTL design is verified using a bit-level hardware model checker, EBMC [6]. The software netlist is verified using a software analyzer, 2LS [190].

Case Study 2: Verification of Buffer allocation protocol

Figure 3.15 gives an example of Verilog RTL (on the left) and the equivalent software netlist in C (on the right). The RTL design has one property. This property is modeled in software netlist using C assertion. The buffer allocation protocol design is safe. The property is proven to be k-inductive for the same values of k in both the Verilog and the software netlist.

An important characteristic of the software netlist design is that it is structurally identical to the Verilog RTL. Thus, it is easier for debugging purposes specially when the designs are continuously evolving.

3.6 Conclusion

This chapter presents a tool for translating Verilog RTL to Software netlist which is expressed in C language. The software netlist representation of the RTL design enables us to leverage technologies from program verification research as well as satisfiability research for formal property verification of hardware at the RTL level. In the subsequent chapter, we explore this direction.

[6] http://www.cprover.org/ebmc/

Verilog RTL

```
module traffic(reset, clk, time_left);
input reset, clk;
output [7:0] time_left;

parameter RED_LIGHT    = 0;
parameter GREEN_LIGHT  = 1;
parameter YELLOW_LIGHT = 2;
wire [5:0] RED_count;
wire [5:0] GREEN_count;
wire [2:0] YELLOW_count;
reg  [1:0] Light_Sign;
reg  [7:0] Counter;

assign RED_count    = 6'h3F;
assign GREEN_count  = 6'h3F;
assign YELLOW_count = 3'h3F;

assign time_left = Counter;

initial begin
 Counter = 0; Light_Sign = 0;
end

always @(posedge clk) begin
 if (reset) begin
  Light_Sign <= RED_LIGHT;
  Counter <= 8'd0;
 end
 else begin
  case (Light_Sign)
   RED_LIGHT :
    Light_Sign <= (Counter == 8'd0) ?
              GREEN_LIGHT : RED_LIGHT;
   GREEN_LIGHT :
    Light_Sign <= (Counter == 8'd0) ?
           YELLOW_LIGHT : GREEN_LIGHT;
   YELLOW_LIGHT :
    Light_Sign <= (Counter == 8'd0) ?
             RED_LIGHT : YELLOW_LIGHT;
  endcase
  case (Light_Sign)
   RED_LIGHT :
     Counter <= (Counter == 8'd0) ?
      GREEN_count : Counter - 8'd1;
   GREEN_LIGHT :
     Counter <= (Counter == 8'd0) ?
     YELLOW_count : Counter - 8'd1;
   YELLOW_LIGHT :
     Counter <= (Counter == 8'd0) ?
        RED_count : Counter - 8'd1;
  endcase
 end
end
assert property (time_left != 8'd255);
// this bug is manifested only
// after 64 iterations
 assert property (Light_Sign != 2'd2);
 assert property (Light_Sign != 2'd3);
endmodule
```

Software netlist

```
struct state_elements_traffic {
  unsigned char Light_Sign;
  unsigned int Counter; };
struct state_elements_traffic straffic;
// parameters
int RED_LIGHT=0;int GREEN_LIGHT=1;
int YELLOW_LIGHT=2;int RED_count=63;
int GREEN_count=63;int YELLOW_count=63;

void traffic(_Bool reset, _Bool clk,
          unsigned int *time_left) {
  unsigned char Light_Sign_old;
  unsigned int Counter_old;
  // assignment statements
  Light_Sign_old = straffic.Light_Sign;
  Counter_old = straffic.Counter;
  if(!reset) {
    straffic.Light_Sign = RED_LIGHT;
    straffic.Counter = 0; }
  else {
   if(Light_Sign_old == RED_LIGHT)
    straffic.Light_Sign = (Counter_old == 0)
    ? GREEN_LIGHT : RED_LIGHT;
   else if(Light_Sign_old == GREEN_LIGHT)
    straffic.Light_Sign = (Counter_old == 0)
    ? YELLOW_LIGHT : GREEN_LIGHT;
   else if(Light_Sign_old == YELLOW_LIGHT)
    straffic.Light_Sign = (Counter_old == 0)
    ? RED_LIGHT : YELLOW_LIGHT;

   if(Light_Sign_old == RED_LIGHT)
    straffic.Counter = (Counter_old == 0)
    ? GREEN_count : Counter_old - 1;
   else if(Light_Sign_old == GREEN_LIGHT)
    straffic.Counter = (Counter_old == 0)
    ? YELLOW_count : Counter_old - 1;
   else if(Light_Sign_old == YELLOW_LIGHT)
    straffic.Counter = (Counter_old == 0)
    ? RED_count : Counter_old - 1; }
 // continuous assignment
 *time_left = straffic.Counter;
}
void main() {
 _Bool reset, clk;
 unsigned int time_left;
 while(1) {
  Traffic_Light(reset, clk, &time_left);
  assert((time_left != 0xffffffff));
  // this bug is manifested
  // only after 64 iterations
  assert((straffic.Light_Sign != 2));
  assert((straffic.Light_Sign != 3));
 }
}
```

Figure 3.14: Verilog RTL (on the left) and software netlist in C (on the right) of traffic light controller circuit

Verilog RTL

```
module BufAl(clock,alloc_raw,nack,
alloc_addr,free_raw,free_addr_raw);
 input        clock;
 input        alloc_raw;
 output       nack;
 output [(4-1):0] alloc_addr;
 input            free_raw;
 input [(4-1):0]  free_addr_raw;
 reg              busy [0:(16 - 1)];
 reg [4:0]        count;
 reg              alloc, free;
 reg [(4-1):0]    free_addr;
 integer          i;

 initial begin
 for (i=0;i<16;i=i+1)
  busy[i] = 0;
  count = 0;
  alloc = 0;
  free = 0;
  free_addr = 0;
 end

 assign nack =
  alloc & (count == 16);
 assign alloc_addr =
 ~busy[0] ? 0 :
       ~busy[1] ? 1 :
       ~busy[2] ? 2 :
       ~busy[3] ? 3 :
       ~busy[4] ? 4 :
       ~busy[5] ? 5 :
       ~busy[6] ? 6 :
       ~busy[7] ? 7 :
       ~busy[8] ? 8 :
       ~busy[9] ? 9 :
       ~busy[10] ? 10 :
       ~busy[11] ? 11 :
       ~busy[12] ? 12 :
       ~busy[13] ? 13 :
       ~busy[14] ? 14 :
       ~busy[15] ? 15 :
       0;

  always @ (posedge clock) begin
   alloc = alloc_raw;
         free = free_raw;
         free_addr = free_addr_raw;
  end
  always @ (posedge clock) begin
         count = count + (alloc & !nack)
          - (free & busy[free_addr]);
if (free) busy[free_addr] = 0;
if (alloc & !nack) busy[alloc_addr] = 1;
  end

  assert property: ((count[4] == 0
              || count[3:0] == 0));
endmodule
```

Software netlist

```
_Bool nondet_bool();
unsigned char nondet_uchar();

struct state_elements_BufAl{
  _Bool alloc;
  _Bool free;
  unsigned char free_addr;
  _Bool busy[16];
  unsigned char count;
};
struct state_elements_BufAl smain;
void initial() {
        int i;
  for (i = 0; i < 16; i=i+1)
          smain.busy[i] = 0;
        smain.count = 0;
        smain.alloc = 0;
        smain.free = 0;
        smain.free_addr = 0;
}
void BufAl(_Bool clock, _Bool alloc_raw,
 _Bool *nack, unsigned char *alloc_addr,
 _Bool free_raw,
 unsigned char free_addr_raw)
{
 unsigned char tmpaddr, tmp;
 // always clocked block
 tmpaddr = smain.free_addr & 0xF;
 smain.count = ((smain.count & 0x1F) +
 (smain.alloc & !(*nack)) - (smain.free
 & smain.busy[tmpaddr]))&0x1F;
 if(smain.free)
  smain.busy[smain.free_addr] = 0;
 tmp = *alloc_addr;
 if(smain.alloc & !nack)
  smain.busy[tmp] = 1;
 smain.alloc = alloc_raw;
 smain.free = free_raw;
 smain.free_addr = free_addr_raw;
 // continuous assignment statements
 *nack = smain.alloc &&
 ((smain.count & 0x1F) == 16);
 *alloc_addr =
 !smain.busy[0]?0:!smain.busy[1]?1:
 !smain.busy[2]?2:!smain.busy[3]?3:
 !smain.busy[4]?4:!smain.busy[5]?5:
 !smain.busy[6]?6:!smain.busy[7]?7:
 !smain.busy[8]?8:!smain.busy[9]?9:
 !smain.busy[10]?10:!smain.busy[11]?11:
 !smain.busy[12]?12:!smain.busy[13]?13:
 !smain.busy[14]?14:!smain.busy[15]?15:0;
}
void main() {
  _Bool clock, alloc_raw, nack, free_raw;
  unsigned char alloc_addr, free_addr_raw;
  // initial block
  initial();
  while(1) {
   alloc_raw = nondet_bool();
   free_raw = nondet_bool();
   free_addr_raw = nondet_uchar();
   BufAl(clock, alloc_raw, &nack,
   &alloc_addr, free_raw, free_addr_raw);
   assert(((((smain.count >> 4) & 1) == 0)
         || ((smain.count & 0xF) == 0)));
  }
}
```

Figure 3.15: Verilog RTL (on the left) and software netlist in C (on the right) of buffer allocation protocol circuit

Chapter 4

Hardware Verification Using Native Software Analyzers

With the ever-increasing complexity of hardware and SoC-based designs for mobile platforms, demand for scalable formal verification tools in the hardware industry is always growing. The scalability and performance of hardware model checking tools broadly depends on three key factors: the *design representation*, the *verification engine*, and the *proof engine*. This chapter experimentally evaluates the influence of the first two factors: the design representation, and the verification engine. In particular, this chapter presents a novel formal verification framework for RTL designs in software languages with native software verification tools. In this chapter and the subsequent chapters of this dissertation, we interchangeably use the terms *software netlist* and *program*.

4.1 Hardware Model Checking

Figure 4.1 gives the phases that a typical hardware model checking tool passes through.

Design representation

Given a hardware design in Verilog RTL, formal verification tools use different internal representations for the design, at different levels of design granularity: *bit level* or *word level*. Figure 4.1 lists some of the design representations commonly used. Most formal verification tools for hardware [31, 35, 38] synthesize the input design into a bit-level netlist, typically represented as *AIG*, and stored in formats such as *BLIF*, *EDIF*, *PLA* or *BAF*. The bit-level netlist consists of a network of and-gates, inverters, and memory elements referred to as registers. This approach misses the opportunity to exploit the word-level structure of the input RTL design. Tools based on word-level representations may use *BTOR* or another intermediate word-level format, which enables the use of word-level decision procedures,

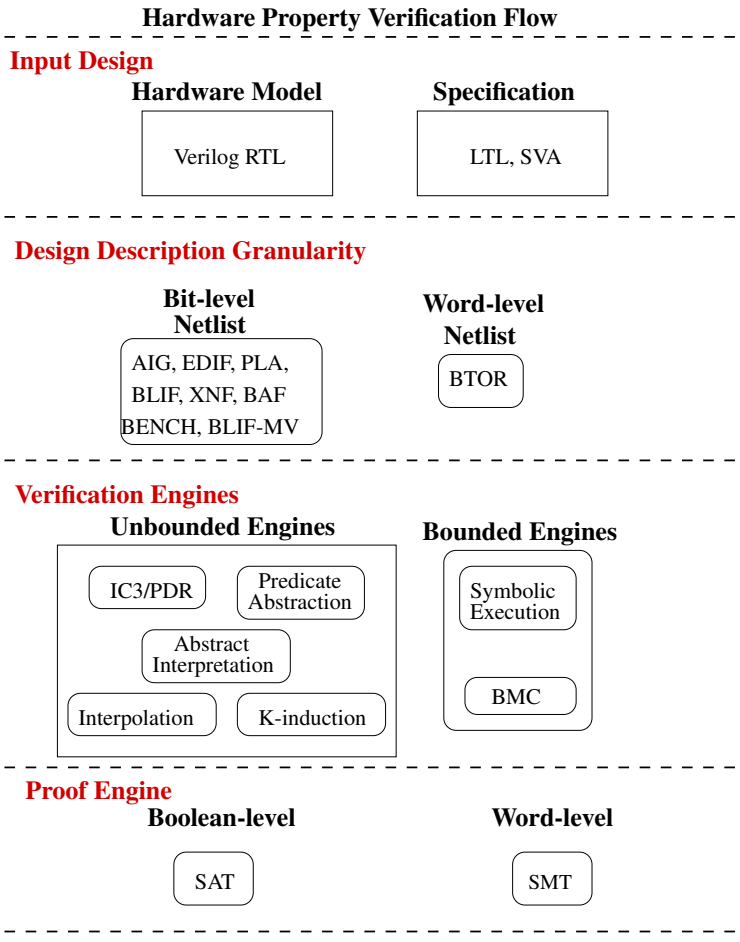


Figure 4.1: Conventional flow for hardware property verification

such as Satisfiability Modulo Theory (SMT) solvers, in the back-end of these tools. The word-level netlist use "words" which are the *bit-vector* encoding of the registers and wires, rather than representing them as individual bits.

The Verification Engine

Over the past few years, we have seen that verification tools participating in Hardware Model Checking Competition (HWMCC) [1] employ a variety of verification engines to speed up proofs or to detect deep bugs. After McMillan's notable work on interpolation-based model checking [158], the work of Bradley [31] on incremental inductive invariant generation (IC3) proved to be a paradigm shift in scaling bit-level hardware verification. The success of IC3 is its ability to perform unbounded verification as well as to quickly detect deep bugs based on relative inductive strengthening. Techniques like k-induction and interpolation compute only one inductive invariant and heavily rely on the underlying reasoning engine

[1] http://fmv.jku.at/hwmcc17/

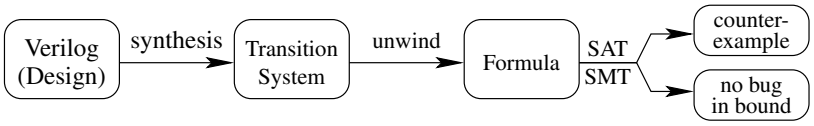


Figure 4.2: Hardware property verification using BMC

(SAT or BDD) because of their unrolling-based strategy. By contrast, IC3 is incremental in nature and computes many inductive invariants that together strengthen the property.

In the past, we have seen the use of high-level structures as a design description for hardware model checking [46, 48, 129, 138, 171, 206]. The efficiency and scalability of the verification engine depends on the granularity of design descriptions as well as the decision procedures used. This is exemplified by the use of word-level hardware verification [143, 206] tools such as word-level STE [46], BMC [24], predicate abstraction [129] and interpolation [145]. The performance of word-level symbolic execution engines are determined by the level of abstraction of the symbolic expressions and the power of the rewrite engine used by the SMT solvers. Recent work described in [52, 218] generalizes Property Directed Reachability (PDR) to richer logics such as Quantifier-Free Bit-Vector logic (QF_BV) and the software domain—and demonstrates better scalability than bit-level PDR.

Consider Bounded Model Checking (BMC) as an exemplar of the way contemporary formal verifiers for hardware work (Figure 4.2). The input Verilog design is first translated into a transition relation at register-transfer level, in which one transition corresponds to one clock cycle. The transition relation for a system and its specification are jointly unwound up to a user-defined depth to form a bit-level or word-level formula. This formula is then given to a suitable SAT or SMT solver. If the formula is determined to be satisfiable, then there is a bug and the verifier extracts a trace of the circuit leading to the bug from the satisfying assignment.

RTL Verification Using Software Analyzers

Unlike traditional approaches that synthesize the design into a bit-level netlist [35, 37] or generate an abstract model of the design (say C/C++ ISA or micro-architectural models derived from RTL [48]), we translate hardware models, articulated in Verilog at register transfer level, into a *software netlist*, represented as ANSI-C program. The generated model is expressed as a software program.

The natural software language for our representation is C, so our approach bears some resemblance to methods for verifying system-level or transaction-level descriptions of hardware written in C/C++ or SystemC.

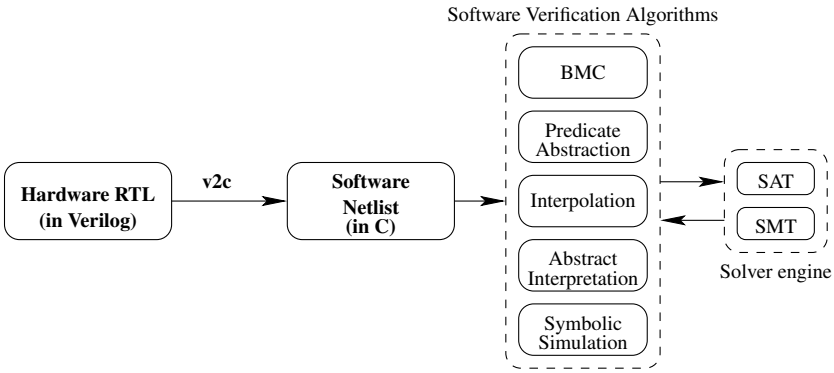


Figure 4.3: RTL verification using software analyzers

The software netlist representation allows us to leverage various software verification technologies including abstract interpretation [70], that have not been applied to RTL verification. An overview of the proposed verification framework as shown in Figure 4.3.

In this chapter, we present an empirical evaluation of various unbounded formal verification algorithms such as Predicate Abstraction, *k*-induction, Interpolation, IC3/PDR and Abstract Interpretation; and we compare the performance of verification tools from the Software Verification Competition (SV-COMP'16) and Hardware Model Checking Competition (HWMCC'16) that use these techniques. Of course, some of these techniques were originally proposed in the hardware community but later used in the software verification community. But, techniques such as abstract interpretation or path-based symbolic execution have never been applied to bit-level hardware verification. Also, this chapter experimentally evaluates *unbounded* formal RTL verification at *bit-level* and using *software netlist*.

One of the obvious differences between hardware verifiction and software verification is that hardware designs tend to use bit-slices in the design. For example, a microprocessor can have a very large instruction word and then slice that word into different parts such as opcode, address, immediate operand, etc. These bit-slices can partition the words and sometimes can even overlap each other. Sometimes, if a design contains too many bit-slices, pure word-level tools can perform poorly compared to bit-level tools. Software verification, in general does not face this problem because bit-slices do not show up in the typical programs.

Claim of novelty

We make the following novel contributions in this chapter.

1. We present a novel verification framework for automatic formal property checking of hardware RTL designs in software language using native software analyzers.

2. By means of experiment, we evaluate various unbounded SAT/SMT based verification techniques, such as *K*-induction, interpolation, IC3/PDR and abstract interpretation for safety verification of RTL and software netlist designs. To this end, we compare the performance of verification tools that employ these techniques from Hardware Model Checking Competition (HWMCC) [2] against software analyzers from Software Verification Competition (SV-COMP) [3].

3. The proposed verification framework allows for direct technology transfer from program analysis research to formal property checking of hardware.

Plan of the chapter

The rest of the chapter is organized as follows. Section 4.2 discusses the related work. The problem statement is described in Section 4.3. Section 4.4 describes the verification methodologies. The equivalence between the hardware and software netlist is explained in Section 4.5. Section 4.6 explains the proposed verification flow. The experimental results are discussed in Section 4.7. Section 4.8 gives a prelude to the materials presented in subsequent chapters. Section 4.9 concludes the chapter.

4.2 Related Work

Technology transfer between the hardware and software verification community over the past two decades have been immensely successful and demonstrated significant drive in development of new verification algorithms in both fields. In 1986, Clarke et al. [56] proposed model checking for finite state concurrent systems. Later, McMillan et al. [61] introduced symbolic model checking in 1996. Biere et al. [23] introduced symbolic model checking without BDDs in 1999. Graf et al. [114] proposed a technique based on predicate abstraction. Later, Clarke et al. [58] used Counterexample Guided Abstraction Refinement (CEGAR) in SMV tool for verifying Fujitsu IP core. Kroening et al. [129] used CEGAR to verify hardware designs written in Verilog RTL. Subsequently, predicate abstraction is used for verifying device drivers in C program by Ball et al. [10]. This lead to the SLAM project by Ball et al. [11] in 2002. In 2000, Stålmarck et al. [192] introduced a technique to check safety properties using induction and SAT solver. Subsequently, induction based approaches are used for automatic analysis of scratch-pad memory code for heterogeneous multicore processors by Donaldson et al. [85], 2010 and for software verification using *k*-induction

[2] http://fmv.jku.at/hwmcc15/
[3] https://sv-comp.sosy-lab.org/2017/

by Donaldson et al. [84] in 2011. In 2003, McMillan et al. [158] introduced Interpolation and SAT-based model checking for verifying commercial microprocessor. Subsequently, Interpolation is used for software verification by McMillan et al. [159] in 2006 and by Kroening et al. [146], in 2011. In 2010, Bradley et al. [30] introduced safety checking by inductive generalizations of the counterexample to induction, a technique commonly known as IC3. Subsequently, IC3 is used for software model checking by Cimatti et al. [52] in 2012. Thus, techniques like predicate abstraction, IC3/PDR, k-induction, and interpolation-based approaches have all made their way from the hardware to the software domain; but the path-wise symbolic execution and abstract interpretation have been primarily used for software verification. A survey of bounded and unbounded SAT-based hardware model checking techniques is presented in [6].

4.3 Preliminaries & Problem Statement

We formally define the safety verification of a transition system T, commonly known as the *hardware model checking problem*. We define a Boolean *transition system*, appropriate for bit-level hardware modeling, as $T = \langle I, x, \delta \rangle$, where $x = \{x_1, x_2, ..., x_n\}$ is the set of state variables, which range over $\mathbb{B} = \{true, false\}$. A *state* s of T is an assignment of values to variables x, the predicate $I(x)$ is a formula over the variables x specifying the *i*nitial state and $\delta(x, x')$ represents the transition relation as a Boolean formula over states x and x', where the primed variables x' represent the next-state value of the variables x. A *trace* $\gamma : s_0, s_1, ...$ is an infinite sequence of states such that $s_0 \models I$, and for each $i \geq 0$, $(s_i, s_{i+1}) \models \delta$. A state s is reachable in T if $\exists \gamma . s \in \gamma$. A safety property P of T is a Boolean formula over the variables x of T, which asserts that certain states s of T cannot be reached during the execution of T. These states are often known as *bad states*, given by the predicate $B(x)$ over the state variables x. We use the symbol $P(x_0)$ to denote that the property P holds in state x_0. The symbol P_k denotes the property at the k-th time frame. We use the symbol δ_k to denote k-th time frame of the transition relation δ.

Problem Statement

Given a state-space over n Boolean variables, the problem is to decide whether $T \models P$, that is, starting from $I(x)$, whether a state in $B(x)$ can be reached following only transitions in $\delta(x, x')$.

Verilog	Formal Semantics
module top(clk, a); **input** clk, a; **reg** b,d,e; **wire** c,cond; **assign** c = e ? 1'b0:d; **assign** cond = a; **always** @(**posedge** clk) **begin** b<=a; **if**(cond && b) e<=b; **else** e<=0; d<=c; **end** **endmodule**	**Combinational Logic** $\forall t,\ c(t) = if\ e(t)\ then\ 0\ else\ d(t)$ $\forall t,\ cond(t) = a(t)$ **Sequential Logic** $\forall t,\ b(t+1) = a(t)$ $\forall t,\ e(t+1) = if(cond(t)\ \wedge\ b(t))\ then\ b(t)\ else\ 0$ $\forall t,\ d(t+1) = c(t)$

Figure 4.4: Verilog RTL and its formal semantics

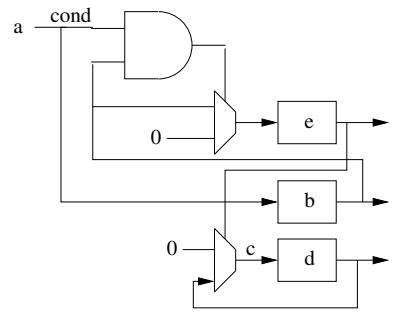
Synthesized Hardware	Software Netlist
	**struct** state_elements_top { **unsigned int** b, d, e; }; **struct** state_elements_top u1; **void** top(_Bool clk, **unsigned** a) { _Bool c,cond; _Bool b_old=u1.b; cond = a; c = (u1.e)?0:u1.d; u1.b = a; **if**(cond && b_old) u1.e = b_old; **else** u1.e = 0; u1.d = c; }

Figure 4.5: Circuit to Software

4.3.1 Circuit to Software

We give an example for translation from Verilog RTL into the software netlist in C. Figure 4.4 gives an example of Verilog RTL on the left and its formal semantics on the right. The RTL has a combinational as well as sequential logic component, whose formal semantics are shown on the right. Note that the $\forall$ quantification used in the formal semantics of Figure 4.4 is defined over time t. Figure 4.5 gives the synthesized hardware representation on the left corresponding to the Verilog RTL design of Figure 4.4. The equivalent software netlist is shown on the right side of Figure 4.5.

4.4 Formal Verification of Software Netlist

In this section, we briefly describe various precise verification algorithms based on SAT/SMT-based decision procedures as well as approximate techniques that operates over relational

and non-relational abstract domains. In this dissertation, we use these techniques for property verification of RTL circuits as well as software netlist designs.

Verification Techniques

Traditional model checkers explicitly compute the postconditions or preconditions to compute the reachable sets of states or the set of states that never lead to the violation of P respectively. Given a transition system T and a property P, we formally describe various well-known algorithms for computing the reachable state space of T such that every reachable state satisfies the property P. We broadly classify these techniques into two separate classes based on whether they use decision procedures or abstract domains – *SAT/SMT-based Verification*, and *Abstraction-based Verification*. For the purpose of illustration, we give bit-level descriptions of all the verification techniques. The materials in Section 4.4.1 and Section 4.4.2 are not new and already exist in the literature.

4.4.1 SAT/SMT-based Verification

We describe few well-known SAT/SMT-based verification techniques which are used for property verification of the hardware RTL circuits as well as the software netlist designs.

Bounded Model Checking

The idea of BMC [22] is as follows. Given a depth k and a set of error states B, BMC operates by unwinding the transition relation T up to depth k starting from initial state x_0, represented by an initial state predicate I. This results in the following formula which is then checked for satisfiability using an efficient SAT or SMT procedure.

$$I(x_0) \wedge \delta(x_0, x_1) \wedge \ldots \wedge \delta(x_{k-1}, x_k) \wedge (B(x_0) \vee \ldots \vee B(x_k))$$

Thus, BMC exploits the finiteness of T and creates k copies of the T for each unrolling.

BMC with K-induction

In practice, BMC technique is incomplete. The k-induction [192] technique is explained as follows. The base case of the k-induction is the simple BMC problem shown before. If the base case is unsatisfiable, then the induction step is checked using the formula shown below.

$$P(x_0) \wedge_{i=0}^{k-1} (\delta_i \wedge P_i) \implies P_k$$

Thus, k-induction assumes that P holds over $k-1$ time steps to increase the likelihood that P holds in k-th time step. However, k-induction requires simple path constraints for completeness [21].

Interpolation-based Model Checking

Interpolation-based model checking [158] computes a step-wise overapproximation of the reachable state-space, Q, for a fixed unrolling k of T, which is shown as follows.

$$Q \wedge_{i=0}^{k-1} (\delta_i \wedge P_i) \implies P_k$$

The value of k is increased when the implication fails. This progressively yields better overapproximation by generating an interpolant between the i-th overapproximation and k-step unrolling. Thus, the interpolation technique computes an abstraction of the postcondition with respect to the property P, by deriving interpolants from the failure of the BMC problem. Interpolation-based model checking is a complete method for reachability analysis of finite-state systems.

Predicate Abstraction with CEGAR

This technique computes an abstract model $\hat{T}$ from the concrete model T using existential abstraction [58, 114]. Given a concrete state x which is the valuation of all registers and a set of predicates, $Pred = \{\pi_i \ldots \pi_k\}$, an abstract state b is obtained by applying an abstraction function α, such that $b = \alpha(x)$. The abstract state b is given by a vector of Boolean values of the predicates in $Pred$. Let S be the set of concrete states in T. We define the abstract model given by, $\hat{T} = \langle \hat{I}, \pi, \hat{\delta} \rangle$. We define the components of $\hat{T}$ and property verification using predicate abstraction below.

1. Transition Relation: $\hat{\delta} \hat{=} \{(b,b') \mid \exists x,x' \in S : \delta(x,x') \wedge \alpha(x) = b \wedge \alpha(x') = b'\}$, $x = \{x_1 \ldots x_n\}, b = \{b_1 \ldots b_n\}, b_i = \pi_i(x)$, π_i is the predicate on concrete variable x_i.
2. Initial State: $\hat{I}(b) \hat{=} \exists x \in S : (\alpha(x) = b) \wedge I(x)$
3. Safety Property: $\hat{P}(b) \hat{=} \forall x \in S : ((\alpha(x) = b) \implies P(x))$. Thus, if property $\hat{P}$ holds on all reachable states of the abstract model $\hat{T}$, then P also holds on all reachable state of concrete model T.

Property Directed Reachability

The property directed reachability [31] technique is briefly described as follows. Let $\alpha_1(x), \alpha_2(x), \ldots, \alpha_n(x)$ be a sequence of inductive assertions generated by the PDR algorithm such that P is an invariant of T if the following conditions hold.

1. $I(x) \implies P$
2. $\forall i, I(x) \implies \alpha_i(x)$

3. $\forall i, \bigwedge_{i=1}^{k-1}(\alpha_i(x) \wedge P(x) \wedge \delta(x,x') \implies \alpha_k(x'))$, α_k is inductive relative to $\alpha_1, \alpha_2, \ldots, \alpha_{k-1}$,

4. $\forall i, \bigwedge_{i=1}^{n}(\alpha_i(x) \wedge P(x) \wedge \delta(x,x') \implies P(x'))$, P is inductive relative to the inductive invariants $\alpha_1, \alpha_2, \ldots, \alpha_n$,

The algorithm computes successive intermediate inductive assertions by removing the counterexamples-to-induction states in property-directed fashion. Thus, the technique computes an overapproximation of the set of reachable states in successive steps until it finds an inductive strengthening assertion sufficient to prove the property.

4.4.2 Abstraction-based Verification

We now describe few well-known abstraction-based verification techniques that analyzes the transition system by computing an abstract pre-condition or abstract post-condition over a given abstract domain. Until now, the techniques described below are purely used in the context of software programs. In this dissertation, we employ these techniques for verifying the software netlist designs generated from the RTL.

Abstract Interpretation

Abstract Interpretation [68, 70, 71], is a theory of sound approximation of program semantics based on lattice structures. Static analysis using abstract interpretation is widely used to verify properties of safety-critical systems. In abstract interpretation, a given program is analyzed with respect to a given *abstract domain D*. The technique computes an abstraction of the postcondition to derive an inductive invariant *AInv*. Elements of an abstract domain can be sets or conjuncts of formulae [33]. Given a transition system T, a set of formulas A each of which is an element of D, and property P as above, *AInv* is computed as follows.

1. $\exists AInv \in A. \forall x_0, x_1 \in T. (I(x_0) \implies AInv(x_0)) \wedge (AInv(x_0) \wedge \delta(x_0,x_1) \implies AInv(x_1))$,

2. $\forall x. (AInv(x) \implies \neg P(x))$

Automata-based Trace Abstraction with Interpolants

Automata-theoretic approaches to program verification [126, 149, 213] construct an automaton that represents a program together with its specification. The negation of the specification is encoded via error locations in the automaton, that is, the error location is an accepting state. The program satisfies the specification iff each trace accepted by the automaton is infeasible. One of the automata-theoretic approaches for program verification is implemented in a tool called *UltimateAutomizer* [125] (winner of SVCOMP'17). The technique in [125]

decomposes a program into sets of traces, and the decomposition is guided by proofs that are obtained for single traces. For correct programs, every non-empty set of error traces is an abstraction of the feasible error traces. Each iteration of the algorithm tries to obtain loop invariants for the path program induced by the current counterexample. This may be done through different techniques which can be applied iteratively. Each technique provides a set of Hoare triples that constitute a proof of safety for the current counterexample. As soon as a technique yields loop invariants, the algorithm uses these loop invariants during the refinement and generalization step of trace abstraction. If all the techniques have been exhausted and a loop invariant has not been obtained yet, the algorithm use some or all (depending on the strategy) of the obtained hoare triples and refine or generalize with them. Note that refinement reduces the number of error traces.

4.5 Equivalence of RTL and Software Netlist

The consistency between two designs can be established in several ways. For example, a few common notions of consistency are *behavioral*, *cycle-accurate*, *non-cycle accurate* or *functional* consistency. The consistency between a reference model and an implementation model is established through systematic equivalence checking [16,141,142,204,212]. Equivalence checking between a timed and untimed model is a hard problem [148]. Sequential equivalence checking techniques are used to check the equivalence between an asynchronous event driven semantics of C and synchronous clock driven semantics of Verilog [16, 142].

Two designs are said to be *cycle-accurate-equivalent* [95, 148] if they produce the same output using the same number of clock cycles. On the other hand, two designs are *not* cycle-accurate-equivalent if they require different cycles to perform the same computation. For example, consider two circuits implementing Euclid's algorithm to compute Greatest Common Divisor (GCD), where Circuit A uses two subtractors and Circuit B uses one subtractor. Assume that Circuit B requires more cycles than Circuit A for computing the GCD. Thus, they are functional-equivalent but not cycle-accurate-equivalent. However, Circuit A and Circuit B can be made cycle-accurate-equivalent by making Circuit A to stutter for few cycles until Circuit B finishes its slower operation. This may require some additional logic in Circuit A to determine the condition for stuttering.

In this dissertation, we consider the equivalence between the RTL circuits and the software netlists that preserve the following - 1) the input-output behavior, and 2) the outcome of all assertions (temporal or non-temporal) must match in the RTL as well as in the software netlist design. We first present the property specification language and then define the property-based observable equivalence.

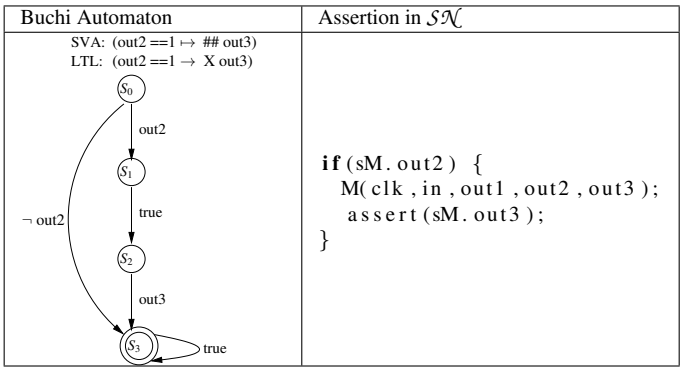


Figure 4.6: LTL and its corresponding C assertion

Properties

System Verilog Assertions (SVA) property specification language is most commonly used to write properties of RTL designs. The SVA properties can be translated into an intemediate Linear Temporal Logic (LTL) [198] representation. LTL is a modal logic that is commonly used to reason about a hardware transition system. The modalities in LTL refer to temporal operators such as G (global), F (eventual), X (next) and U (until). An LTL formula may contain propositional symbols, boolean or temporal operators. We refer the reader to [9, 198] for more details on LTL.

The translation of LTL to a *Buchi Automaton* (BA) is a well-known technique [101, 201] and has been successfully used for LTL model checking, such as the Spin model checker [128]. Further, a Buchi Automaton can be modeled using a C program that simply encodes its state transition relation. In this dissertation, we use System Verilog Assertions (SVA) to express properties of the RTL designs. The SVA properties are translated into equivalent C assertions either using *v2c* or manually. An alternative option, which we did not explore in this work, is to construct a property compiler for automatically translating the SVA properties to assertions in C language.

Figure 4.6 gives an example of SVA, its equivalent LTL, and the corresponding assertion in the software netlist for the design of Figure 4.7. The next time operator X in LTL is modeled by invoking the top-level procedure $M(clk, in, out1, out2, out3)$ in the software netlist. Note that the top-level procedure in the software netlist corresponds to the top-level module of the RTL design.

Property-based Observable Equivalence

Recall that a RTL design in Verilog is translated into a software netlist in C for the purposes of formal verification. To this end, the formal specification or properties (temporal or non-

temporal) over RTL signals that capture hardware behaviors are translated into assertions over the corresponding software variables in the software netlist and checked for consistency against the software netlist. Thus, our notion of equivalence is based on the signals that are observable in the properties under consideration. We call this equivalence criteria *property-based observable equivalence*. Intuitively, this means that the state of latches and wires referred to by the property must match with the corresponding variables in the software netlist at some designated clock cycle. Let us consider the temporal assertion given by, `a |-> ##4 b`. Formal verification tools for hardware unwinds the RTL transition system 4 times for checking the property. Here, unwinding means duplicating the transition system 4 times. Similarly, the software netlist is also unwound 4 times for checking the property. Hence, the simulation of the RTL clock is achieved by unwinding the transition system of the software netlist design. By means of experiment, we show that for designs considered in this dissertation, that is, for single-clock designs without clock-gating or power-gating, abstracting the RTL clock in the software netlist does not have any impact on the verification outcome. However, designs that exhibit clock-gating or power-gating techniques [139] or designs that contain multiple clocks may be non-trivial to translate into the software netlist and may require modeling the behavior of clock explicitly.

Definition 4.5.1. (Property-based Observable Equivalent) Two designs C_1 and C_2 are *property-based observable equivalent* with respect to a property P if the following conditions hold. Note that we assume that the verification outcome of these properties are obtained using the native verifiers for software and hardware. The verification outcome can be either *safe* in which case P holds on C_1 and C_2, or *unsafe* in which case P does not hold on C_1 nor on C_2.

1. For a non-temporal property P, the verification outcome of P must match in $C1$ and $C2$ at every clock cycle.

2. For a temporal property P, the verification outcome of P must match in C_1 and C_2 at clock cycles determined by the temporal operator used in P. We explain this below.

Two designs C_1 and C_2 are *not* property-based observable equivalent with respect to a non-temporal property P if there exist a clock cycle where the verification outcomes of P in C_1 and C_2 do not match. Intuitively, this means that at some cycle N, the state of latches or wires that are referred to by P, does not match in C_1 and C_2, resulting in inconsistent verification outcome for P. On the other hand, two designs C_1 and C_2 are *not* property-based observable equivalent with respect to a temporal property P if there exists a *designated* clock cycle where the verification outcomes of P in C_1 and C_2 do not match.

Example 4.5.1. Figure 4.7 gives an example of a Verilog RTL design (on the left) and the corresponding software netlist in C (on the right) that are property-based observable equivalent with respect to a temporal property (marked in blue). The property in the Verilog RTL design is specified in SystemVerilog Assertion (SVA) [91] language. The equivalent assertion in the software netlist is shown on the right (marked in blue).

Figure 4.8 gives the waveform view that shows the behavior of the RTL design of Figure 4.7. We explain the equivalance of the Verilog RTL and the software netlist design with respect to the temporal SVA assertion given by `(in==1 |-> ##1 out3)`.

Suppose `in=1` at clock cycle i. Then, the state of the latch `out3` is not consistent in Verilog and C at cycle i. This is due to the fact that the value of `out3` *stabilizes* one cycle after the input `in=1` was set, that is, `out3` is stabilized in cycle $i+1$. This behavior is formally captured in SVA by the expression `(in==1 |-> ##1 out3)`. The SVA property specifies that the latch `out3` is high exactly one cycle after `in` was high.

Recall from Definition 4.5.1 that for two designs to be property-based observable equivalent with respect to the temporal property P, the verification outcome of P must match in the two designs at some designated clock cycle. For the above SVA assertion, the designated clock cycle is specified by `##1` delay operator. That is, the value of `out3` must match in Verilog and the software netlist one cycle after the event `(in==1)` has occured. Thus, the value of `out3` may be inconsistent in Verilog and software netlist in clock cycle i when the event `(in==1)` has occured, but the property still holds in both the designs since the output value of `out3` matches in clock cycle $i+1$. We use a state-of-the-art hardware model checker and a commercial software analyzer to verify the RTL and the software netlist respectively. The above SVA property is proven equivalent for an unbounded number of cycles in both the designs. Hence, both the designs are property-based observable equivalent.

On the other hand, the temporal operator *eventual* is handled in a way that the property defines a series of evaluations but is successful when at least one of those evaluations succeed. Further, the temporal operator *until* in a property, say `a until b`, evaluates to true iff `a` is true at the initial evaluation and continues to hold true until, but not necessarily including, a evaluation at which `b` is true. The past-time temporal logic [137] allows past-time operators to be evaluated on the past values of signals and can be handled accordingly. However, the past-time temporal operators are not part of SVA, hence they are not used in this dissertation. Techniques such as translation validation [180] that check the consistency between an RTL and software is a different problem and beyond the scope of this work. However, experiments have shown that for property verification, valid safety properties are proven to be k-inductive for the same unwind depth k in the RTL and the software netlist

Verilog RTL

```
module M(clk, in,
     out1, out2, out3);
input clk,in;
output reg out1,
     out2, out3;
wire t1;

initial begin
out1=0;
out2=0;out3=0;
end

assign t1=out1;

always @(in) begin
out1 <= in;
end

always@(t1) begin
out2 <= t1;
end

always@(posedge clk)
begin
out3 <= out2;
end
endmodule

module main(clk);
input clk;
wire in,out1,
     out2,out3;
M m1(clk,in,out1,out2,out3);
assert property
 (in |-> ##1 out3);
endmodule
```

Software netlist (in C)

```
struct state_M {
 bool out1, out2, out3;
};
struct state_M sM;

void initial() {
 sM.out1=0; sM.out2=0;
 sM.out3=0;
}

void M(bool clk, bool in,
 bool *out1, bool *out2,
 bool *out3) {
  bool t1;
  sM.out1=in;
  t1=sM.out1;
  sM.out2=t1;
  sM.out3=sM.out2;
  // update output
  *out1=sM.out1;
  *out2=sM.out3;
  *out3=sM.out3;
}

int main()
{
 bool clk,in,
 out1,out2,out3;
 initial();
 while(1) {
  if(in) {
   M(clk,in,
   &out1,&out2,&out3);
   assert(sM.out3);
  }
 }
}
```

Figure 4.7: Property-based observable equivalence of RTL and Software netlist

design. Conversely, for unsafe designs, a bug is found in the same unwind depth for both the designs.

4.6 Formal Hardware Verification Tool Flow

Figure 4.9 gives the tool flow for hardware RTL verification using bit-level netlist, word-level netlist and software netlist. The bottom flow of Figure 4.9 shows the bit-level verification flow using ABC. ABC does not support Verilog, so we use an open-source synthesis tool, YOSYS 0.5[4] to translate Verilog RTL to BLIF or AIGER [5] which is then passed to ABC for verification. The middle flow of Figure 4.9 shows the bit/word-level verification using the tool EBMC [6]. EBMC supports IEEE 1364.1 Verilog 2005 standards. EBMC synthesize

[4]http://www.clifford.at/yosys/

[5]http://fmv.jku.at/aiger/FORMAT-20070427.pdf

[6]www.cprover.org/hardware/ebmc

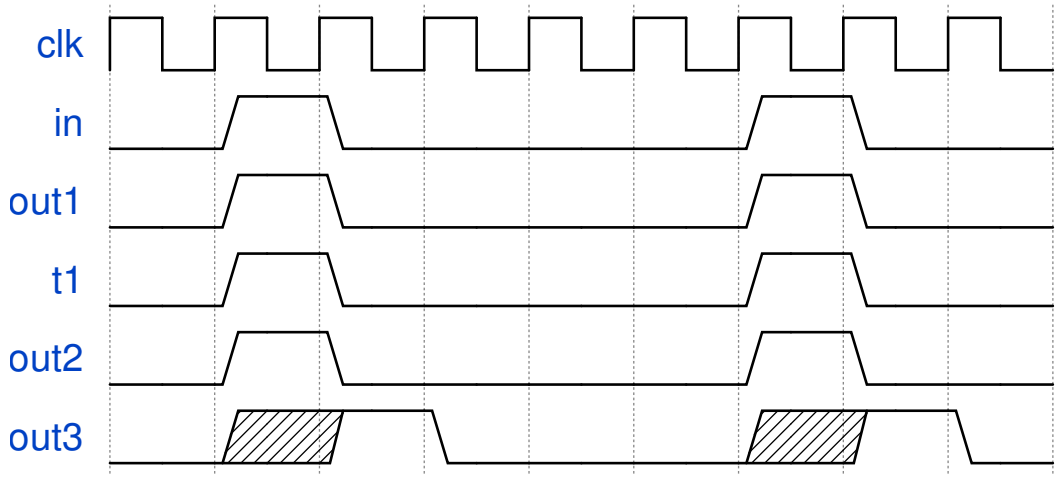


Figure 4.8: Waveform view showing behavior of RTL design in Figure 4.7

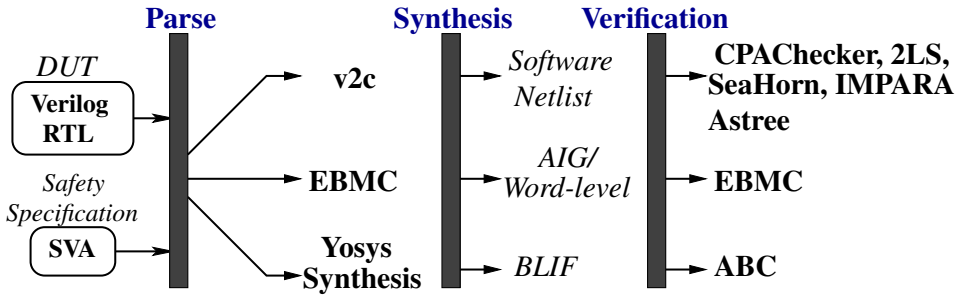


Figure 4.9: Tool flow for hardware property verification

the design in Verilog into a bit-level netlist represented in AIGER or a word-level netlist represented in format that resembles the SMT-LIB standard. The word-level synthesis flow in EBMC is not mature yet. So, we only use the bit-level flow in EBMC. The top flow of Figure 4.9 shows the verification of the software netlist designs. A wide range of representative software verification techniques is applied to determine the safety of the software netlist. In particular, we use *k*-induction [192] (implemented in the tools CBMC [54] and 2LS [34]), interpolation (CPAChecker [20], IMPARA [216]), Abstract Interpretation (Astŕee [76]), IC3/PDR (SeaHorn [121]) and Automata-theoretic verification (UltimateAutomizer [125]).

4.7 Experimental Results

In this section, we report experimental results for *unbounded* safety verification of hardware RTL. Our experimental contributions are two folds.

1. Compare off-the-shelf formal verification tools for RTL verification using bit-level

netlist and software netlist. To this end, we compare state-of-the-art hardware model checking tools, such as *ABC 1.01* (winner of HWMCC'15) and EBMC 4.2 [7], with various software analyzers from SV-COMP 2016, such as ULTIMATEAUTOMIZER 3292EADE (winner of SV-COMP'16), CPACHECKER 1.4, SEAHORN (REVISION 07666C810D), 2LS 0.3.4, and a commercial abstract interpretation tool, *Astrée* (version 14.10).

2. Compare various unbounded verification engines such as *k*-induction, interpolation, IC3/PDR and abstraction interpretation that are employed by verification tools from hardware and software domains.

4.7.1 Benchmark and Tool Distribution

The tool binaries for ABC, EBMC, 2LS, CPACHECKER, SEAHORN and ULTIMATEAUTOMIZER along with the scripts for running YOSYS 0.5 and other tools are available at `http://www.cprover.org/hardware/date2016/`.

Benchmarks

We verified a total of 35 circuits given in Verilog RTL. Out of 35, 27 are *safe* benchmarks and 8 are *unsafe*. The benchmarks are derived from real world hardware benchmark suites, including VIS Verilog models [35], the Texas-97 benchmark suite [8] and Opencores [9]. All the benchmarks are available in three different formats (Verilog, AIGER and ANSI-C) at `http://www.cprover.org/hardware/date2016/`.

Safety Properties

The safety properties are specified as SVA language. The SVA properties are translated into C assertions and instrumented in the software netlist design. Below, we present few examples of the concurrent assertions in SVA and corresponding assertions in the software netlist design.

Figure 4.10 and Figure 4.11 gives the SVA property on the left and the corresponding assertions in software netlist on the right for the vending machine benchmark and Huffman encoder/decoder benchmark, respectively. For the purpose of illustration we give partial code fragments for the software netlist designs.

Example 4.7.1. *Assert_1*: *The balance is never negative and never reaches 15.*

[7] www.cprover.org/hardware/ebmc

[8] https://embedded.eecs.berkeley.edu/research/vis/texas-97

[9] http://opencores.org/

SVA	Assertion (in C)
p1: assert property (@(**posedge** clk) vending.total[3]==0 && !(vending.total[4:0]==15));	assert((((vending.total >> 3) & 0x1) == 0) && !(vending.total & 0x1F == 15));

Figure 4.10: Modeling concurrent SVA in software netlist

Example 4.7.2. *Assert_2*: *When a new transmission begins, the decoder is ready in the next clock.*

SVA	Assertion (in C)
p2: assert property (@(**posedge** clk) encoder.shiftreg[9:1] == 1 |-> ##1 decoder.leaf == 1);	**if**(((encoder.shiftreg >> 1) & 0x1FF) == 1) { // call to top level // module of design huffman(clk,addr); assert(decoder.leaf == 1); }

Figure 4.11: Modeling temporal properties in SVA in software netlist

Discussion of the results

We classify the analysis results into two categories 1) precise tools that do not use any abstraction and performs precise reasoning using SAT/SMT solvers for detecting counterexamples or generating invariants (see Section 4.7.2). 2) Abstraction-based tools that performs approximate analysis without using SAT/SMT solvers, that is, using various abstract domains for proving correctness (see Section 4.7.3). Our experiments were performed on an Intel Xeon machine running at 3.07 GHz with 8 cores and 32 GB RAM.

4.7.2 Analysis using Precise tools

Figures 4.12–4.14 report the comparison between various unbounded SAT/SMT-based verification techniques for verifying bit-level netlist and software netlist designs. The plots in these figures report the runtimes of 12 representative circuits that are most difficult to solve out of a total 35 circuits. The timeout was set to 5 hours and memory resources were restricted to 32 GB RAM per benchmark. We categorize the unbounded approaches into three classes:

- *k*-induction (Figure 4.12)
- interpolation (Figure 4.13), and
- PDR together with other hybrid techniques (Figure 4.14).

By hybrid techniques, we refer to predicate abstraction as implemented in CPACHECKER and a combination of *k*-induction, BMC and abstract interpretation as implemented in

2LS [190]. On the x-axis is the analysis time in seconds and on the y-axis we list the benchmarks. The vertical red lines on the right-hand side of the diagrams show timeouts, out of memory, inconclusive (unknown) results, errors (crashes), and wrong results (tool bugs) reported by the tools. The tools can be distinguished by the size of the circles as well as by colour.

Analysis using k-induction

For safe benchmarks, the results of bit-level verifiers and software analyzers are comparable when the properties are 1-inductive or 2-step inductive. However, for complex safety properties, ABC and software analyzers either timeout or took a long time to terminate. We investigated the reason for higher verification times for some safe benchmarks, such as the FIFO controller, the Read Copy Update (RCU) protocol, and Buffer Allocation protocol. We observe that the properties are not k-inductive for sufficiently large values of k, e.g. (k=1000) and thus tools based on k-induction either timeout or took long time to compute the inductive invariant sufficient to prove the property.

For the unsafe benchmarks, for example, DAIO and the traffic light controller, where the bugs are manifested only at 64 and 65 clock cycles respectively, the verification times using ABC and EBMC's k-induction engine are comparable to CBMC and 2LS. Figure 4.12 reports the time taken by the k-induction engine in ABC, EBMC, CBMC and 2LS. We did not report the time for CPACHECKER since the results suggest that its k-induction engine is not as mature yet.

Analysis using Interpolation

Figure 4.13 reports the time taken by the interpolation engine in ABC, IMPARA and CPACHECKER. ABC is the fastest in 9 out of 12 designs. However, it timed out on three complex benchmarks, RCU, FIFO and BufAl. The software interpolation tool, IMPARA, solved only 3 benchmarks out of which one is the complex FIFO design; yet IMPARA either timed out or ran out of memory for the remaining designs. CPACHECKER solved 5 out of 12 cases. None of the interpolation engines were able to prove RCU and BufAl within the time limit.

Analysis using Hybrid techniques

Figure 4.14 reports the time taken by the IC3/PDR engine in ABC, SEAHORN and other hybrid techniques as implemented in CPACHECKER and 2LS. ABC is the clear winner here; it is the only tool that proves the safety of FIFO and BufAl safe within the given

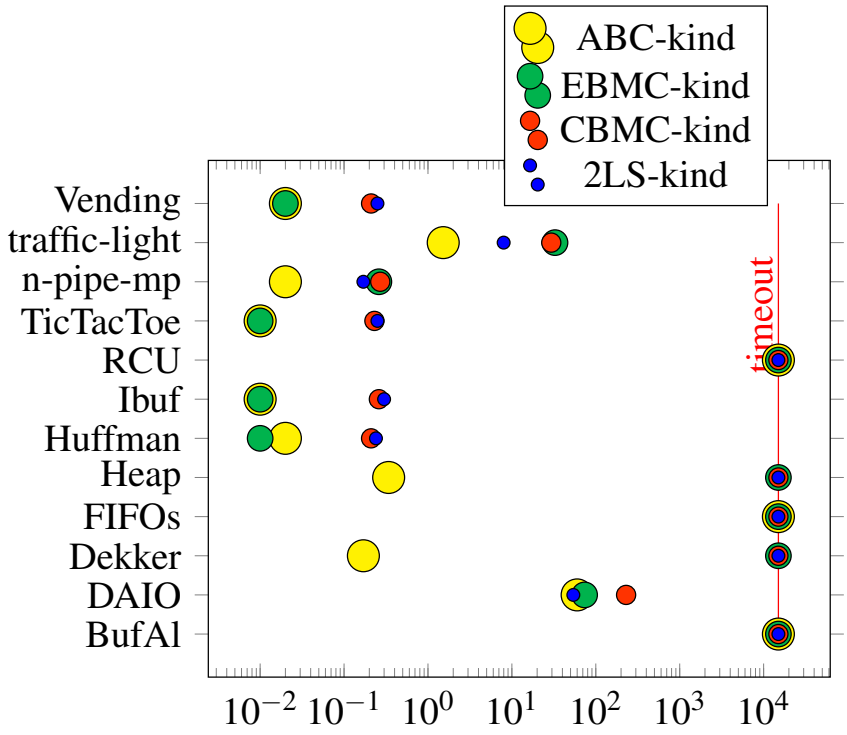


Figure 4.12: Comparison of *k*-induction tools

5h timeout. SEAHORN's PDR engine solves half of the benchmarks but produces false negatives on the other half due to limited support for bitvectors. 2LS successfully solved 8 benchmarks and timed out on 4 benchmarks. CPACHECKER's predicate abstraction reliably solves 7 benchmarks, but timed out on two benchmarks and reports three wrong results. Note that none of the tools were able to prove the safety of RCU design.

Bit-level versus Word-level Analysis

Figure 4.15 gives the runtime comparison between bit-level model checking using SAT solvers and word-level model checking using SMT solvers. The SAT (in DIMACS format) and SMT (in SMT-LIB2 format) benchmarks are obtained automatically using CBMC from the software netlist design. For bit-level results, we report the best runtimes among MiniSAT [1] and Lingeling [2] solvers. The word-level results are obtained using Boolector [3] SMT solver which solved more instances than Z3 [81] SMT solver. The timeout was set to 200 seconds per benchmark. The runtimes in Figure 4.15 are obtained for a pre-determined unwind bound of 70. The bit-level engine in CBMC wins consistently over the word-level engine, except for the 5 cases where Boolector was faster than the bit-level solvers. We investigate the reason for the higher runtimes of the word-level engine. We observe that the RTL designs used for our experiment contains too many bit-slices, which is

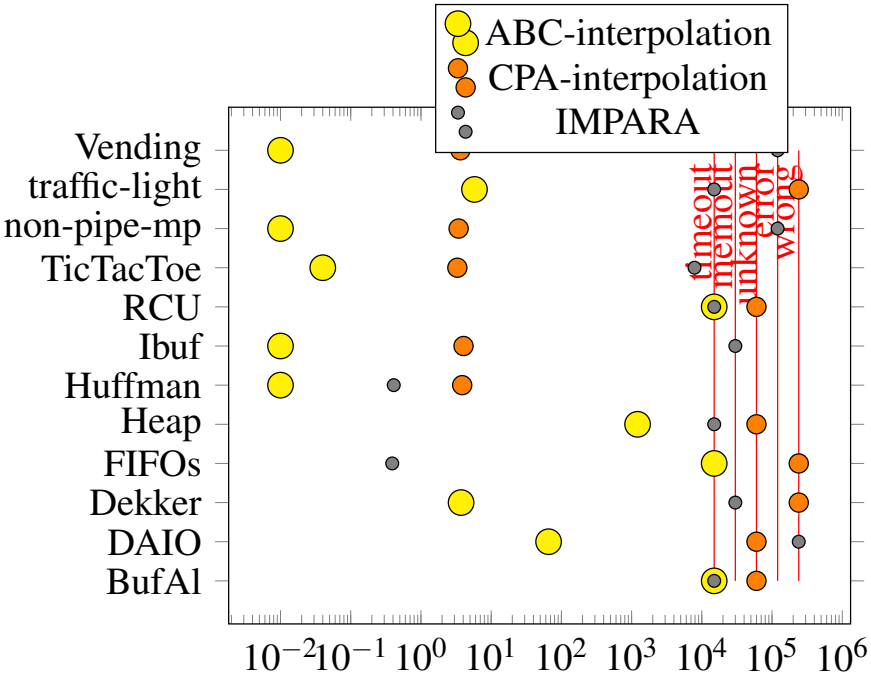


Figure 4.13: Comparison of interpolation-based tools

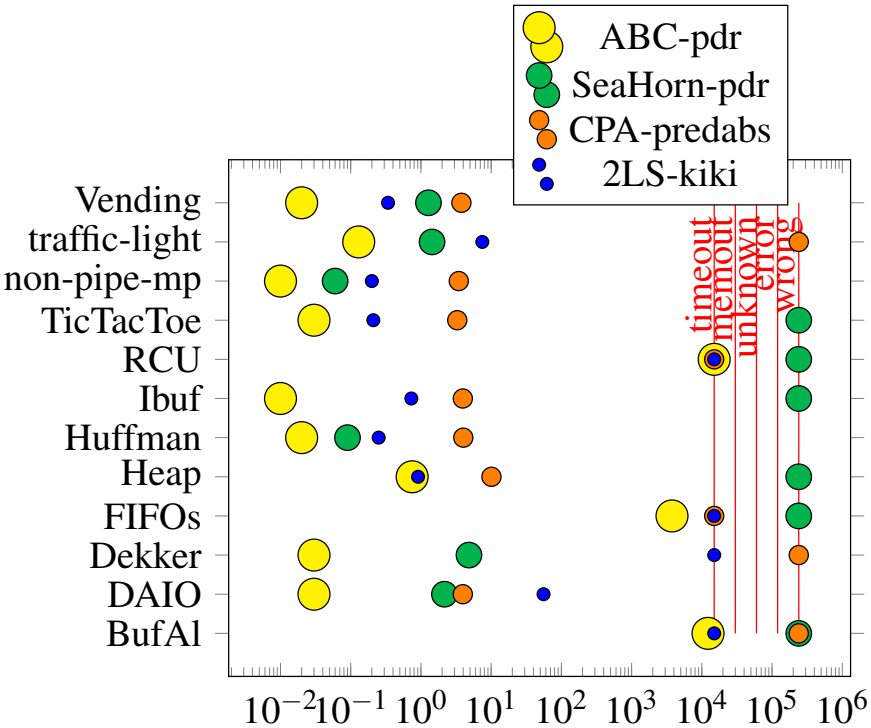


Figure 4.14: Comparison of hybrid techniques

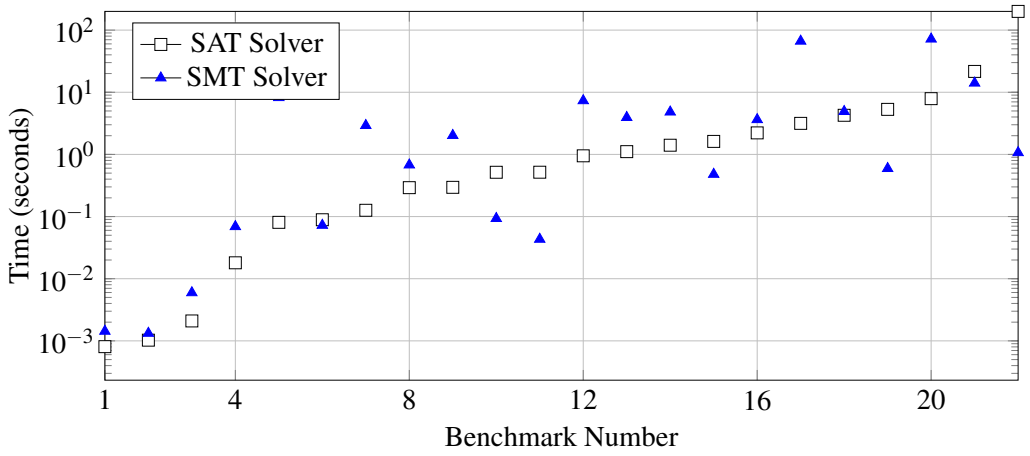


Figure 4.15: Runtime comparison between bit-level and word-level analysis using CBMC

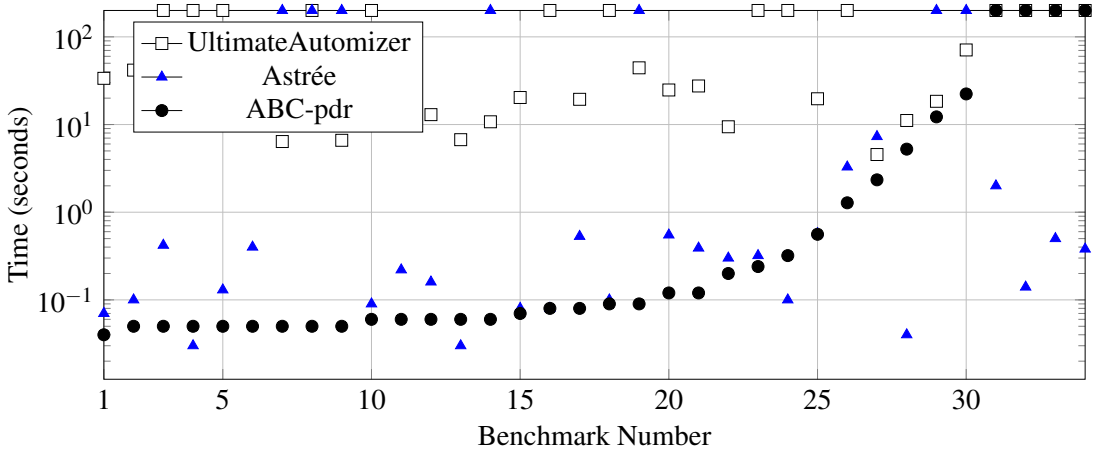


Figure 4.16: Runtime comparison between ABC, UltimateAutomizer and Astrée

also exhibited by the software netlist designs due to the bit-precise translation. Hence, pure word-level reasoning performs poorly compared to the bit-level analysis since word-level tools also use bit blasting for solving such designs.

4.7.3 Analysis using Abstraction-based Tools

We now discuss the analysis using abstraction-based tools. We use the software analyzers Astrée and ULTIMATEAUTOMIZER for this purpose. Recall that ULTIMATEAUTOMIZER implements automata-theoretic approach for program verification. Astrée [76] is a commercial abstract interpretation tool developed by AbsInt. [10] It is primarily used for static analysis of safety-critical software such as avionics software [29]. Astrée employs several numeric abstract domains, such as interval domain that abstract variable values as ranges, as well as

[10] https://www.absint.com/

relational domains that infer linear and non-linear relationships between variables. These abstractions are well-suited for programs performing integer and floating-point arithmetic, but less so for bit-level Boolean operators, including bit-shifts and bitfields packing several values in words. The non-linear abstract domains in Astrée are mostly useful for aerospace applications [18]. Moreover, Astrée features BDD-based abstract domains [19] that can represent non-convex invariants, but they are currently limited to Boolean variables and cannot perform bit blasting on integer variables. For bit manipulating programs generated by V2C, Astrée uses default abstract domains for integer values such as the Interval domain, the Octagon domain, Integer congruences, Integer bitfields, and finite sets (of possible values).

When Astrée encounters an `assert`, it first checks whether there are some program states that do not satisfy the assertion, and then continues the analysis with only the states that satisfy the assertion. Indeed, it assumes that the states that do not satisfy the assertion cause the program to stop. A message such as "Definite assertion failure" appears in case there are no states satisfying the assertion at all, hence, the instructions following the assertion are never executed.

Figure 4.16 gives the runtime comparison between ABC, ULTIMATEAUTOMIZER and Astrée. We choose the ABC PDR engine for comparison with abstraction-based software analyzers since it performs better than the other verification engines and is superior than EBMC on our benchmarks. The tools can be distinguished by the shape as well as by colour. Note that the portfolio engine in ABC-superprove runs multiple verification engines in parallel, but software tool such as ULTIMATEAUTOMIZER and Astrée support only single verification engine. So, for fair comparison, the software tools are compared with the ABC PDR engine only. We verified a total of 35 circuits given in Verilog RTL. The tools ULTIMATEAUTOMIZER and Astrée are fed with the software netlist designs generated from the RTL. On the other hand, the input to ABC is the AIGER representation of the RTL. Both ULTIMATEAUTOMIZER and Astrée operate on an overapproximate abstraction of the concrete program. The timeout was set to 200 seconds and memory resources were restricted to 32 GB RAM per benchmark.

ABC proved a total of 31 benchmarks and timed out for 4 benchmarks. On the other hand, Astrée reported a total of 7 false alarm out of 35 benchmarks (shown by the blue triangles corresponding to the timeout axis in Figure 4.16). However, Astrée was faster than ABC on 8 benchmarks (23% cases). Among these 8 benchmarks, 3 are unsafe for which Astrée reported bugs earlier than ABC. For the remaining 5 benchmarks, ABC timed out in 4 cases. By contrast, Astrée proved all 5 benchmarks within the time limit (see the plots on the extreme right side of Figure 4.16). Overall, the runtimes of Astrée are very close to ABC for most benchmarks.

We investigate the reason for the false alarm in Astrée and observe that the bit manipulating nature of the software netlist prevents Astrée from inferring the necessary invariants since these invariants are not expressible by the underlying abstract domains. On the other hand, ULTIMATEAUTOMIZER timed out in 14 out of 35 benchmarks. For safe benchmarks that are hard to prove, ULTIMATEAUTOMIZER was able to infer the necessary invariants required to prove the property within the time limit, which are shown by square boxes between benchmark number 20 to 35. By contrast, Astrée performed better than ULTIMATEAUTOMIZER for all unsafe benchmarks. Astrée reported bugs in all 8 unsafe benchmarks within the time limit. We now discuss various techniques to improve the precision of analysis using Astrée for the purpose of precise safety verification of the software netlist designs.

Handling Imprecision in Astrée

Typically, most abstract domains in Astrée are non-distributive domains, so they can only represent convex (non-disjunctive) numeric properties. Hence, control-flow joins and loop widening are two major cause of precision loss. Astrée partially alleviates the problem through BDD-based domains and trace-partitioning [182] domain which provides a level of path-sensitivity. To prevent scalability issues, these costly techniques are enabled locally, through automatic heuristics or user guidance. That is, a partitioning initiated inside a function will be merged over before returning from the function; so, it is *function-local*. Sometimes, a longer partitioning is useful, but the heuristics do not generate it (although it can be generated by hand). While standard widening is a technique to accelerate convergence and thus can cause lots of precision loss, Astrée uses more gradual widening and does not loses all precision at once. We employ widening with thresholds, for instance, and also delay the widening (independently on each variable).

With trace partitioning, Astrée automatically insert `__ASTREE_partition_control` directives according to a set of heuristics. Astrée does not partition "everything" since this would be too expensive in practice. Such a high precision is also normally not required for a runtime error analysis. Astrée normally starts with existing partitioning heuristics and the user manually adds partitioning directives in places where false alarms occur due to a loss of precision resulting from merging data-flow information of several control-flow paths (traces). There are two partitioning strategies in Astrée: control-flow partition, and partition over relevant variable values. Both can be either specified by hand or using automatic partitioning or (more likely) a combination of both. Automatic partitioning is simply a fast pre-analysis that inserts `__ASTREE_partition` directives into the Abstract Syntax Tree.

A classic example of control partitioning is to analyze each branch of a control-flow path separately so as to prevent joins at the control-flow merge point. An example of variable

partitioning is as follows. Consider a variable named `mode` that can assume values between 1 and 5. A partitioning directive `__ASTREE_partition_begin((mode));` tells Astrée to keep all five traces for all five possible values of `mode` apart until the next merge point (i.e., the end of a function, a loop, a sub-statement or a merge directive). This, of course requires Astrée to know that mode is in the range $[1,5]$.

Although one can theoretically get an arbitrary high precision in Astrée by partitioning such that all relevant execution paths are considered separately, but that is not feasible in practice considering the analysis times. Hence, the trick is to find the "right" partitioning strategy that is as imprecise as possible but can still prove the property under verification. But this is a challenging task. However, Astrée can show the user any variable range and invariants that it has computed. So, the user can use this information to add manually more partitioning and re-run the analysis. What Astrée lacks is a way for this information to be used automatically in partitioning strategies during the subsequent analyses. For our benchmarks, we inspected all the invariants inferred by Astrée and added the necessary partitioning. A few false alarms in Astrée are avoided using this strategy. However, this requires manual intervention to guide the tool to precisely prove properties of the software netlist design.

Soundness of Astrée

Astrée is sound. That is, it does not miss any genuine error. Astrée found genuine bugs in all 8 unsafe benchmarks. However, it errs for the safe benchmarks, sometimes too much. Hence, Astrée may report more false alarms than other tools.

4.7.4 Summary of the results

We summarize the results obtained from the precise and abstraction-based software analyzers for verifying the software netlist designs.

A fundamental difference betwen the hardware and software designs is that hardware designs tend to use bit-slices to model various functionality including power, performance logic. The software netlist models the bit-level hardware logic using combinations of bit-wise operations, such as shift, bitwise-AND, bit-masking etc. This often makes the software netlist designs hard to analyze as looking at shift and AND operators, in general, would be difficult to reverse them to their original bit-slice intention of the RTL. A common solution is to partition the original word to adapt to the user defined bit-slices. However, this approach would not be feasible since the software netlist design is modeled in the C language, which

supports word-level variables of fixed width (e.g. 32 bits or 64 bits). So this is inherently a problem that is not addressed by software analyzers.

Analysis using Precise SAT/SMT based Software Tools

Though the invariant inference techniques employed by precise software analyzers such as CPACHECKER and 2LS have never been optimized for hardware analysis, the results in this chapter show that these tools are within one order of magnitude compared to bit-level hardware model checkers for detecting bugs as well as proving safety of some of the software netlist designs. However, running SAT/SMT based software analyzers on the software netlist designs exhibits many timeouts, wrong results and errors. We observed that software netlists heavily use bit-level operations and thus bit-precise reasoning ability is necessary to precisely reason about these designs. But, bit-level operations are less prevalent in conventional software and hence less tested in software analysis tools. Thus, out of 35 benchmarks, IMPARA reported 1 wrong result, CPACHECKER reported 3 tool bugs in its interpolation engine, and 3 bugs in predicate abstraction engine and SEAHORN reported 5 tool bugs in its PDR engine due to limited support for bitvectors. We reported the tool bugs to the authors of these tools.

Analysis using Abstraction-based Tools

The abstraction-based software analyzers such as Ultimate Automizer and Astrée performed better than the SAT/SMT-based software analysis tools. Though Astrée requires manual intervention to select the right set of partitions of program traces in order to do precise analysis, the verification runtimes of Astrée are comparable to ABC for most of the benchmarks. However, Astrée often uses numerical abstractions, which are likely to lose important bit-precise information. As a consequence, many inductive invariants required by the bit-manipulating benchmarks cannot be represented, and so not inferred, using current abstractions in Astrée. An abstract domain for this purpose must keep precise relations between the bits of an integer (possibly some form of BDD on the bits of the variable). Astrée has already been refined for avionics and space software by adding few domain-specific abstract domains which were very effective in removing some classes of false alarms [18]. We thus believe that there is scope here for new abstract domains targeting hardware designs while still remaining within the scope of the traditional abstract interpretation. In this dissertation, we present an alternative solution to refine abstract interpretation using satisfiability procedures, which is described in Chapter 6.

Limitations of the Result

We discuss some of the limitations of our proposed approach. Firstly, the publicly available Verilog RTL designs often lack requirement specifications. Hence, it is difficult to write meaningful properties for exhaustive verification of such designs. However, we verified at least one functional safety property for each design. Secondly, the benchmarks used for our experiment in this chapter are limited to block-level designs. Thirdly, most of the properties that we verified are either global properties or temporal bounded properties, that is, they are either of the form $G(x)$, where x must be *true* globally or is of the form $x \mapsto \#\#[1:3]y$, where an event y is asserted between 1st and 3rd timeframe after the event x has happened. We did not verify properties with *eventual* or *release* operators. Fourthly, the proposed approach is only used for the verification of the single-clock RTL designs that does not employ clock-gating or power-gating techniques [139]. Hence, abstracting the RTL clock in the software netlist does not have any impact on the verification outcome in our experiments. However, designs that exhibit clock-gating or power-gating techniques or designs that contain multiple clocks may be non-trivial to translate and may require modeling the behavior of clock explicitly in the software netlist, which is shown in Section 3.3.2.

4.8 Prelude to Abstract Conflict Driven Learning for Safety

This chapter presents findings from applying various native software analyzers to RTL verification. We learned that standard static analysis using abstract interpretation is efficient but imprecise. On the other hand, precise SAT/SMT-based model checkers that uses bit blasting are precise but inefficient. The imprecision in abstract interpretation is due to the control-flow joins, loop widening or use of the complex bit-manipulating operations in the software netlist design. We also learned that some degree of imprecision in static analysis tools can be reduced by doing aggressive manual partitioning of the software netlist design.

In this dissertation, we present a technique for precise abstract interpretation of the software netlist designs using CDCL-style analysis that automatically partitions the traces of the software netlist in a way that the technique can prove correctness for each partition. To be efficient, the technique infers partitions that are just precise enough. To this end, Chapter 6 present an abstract interpretation framework that generalizes CDCL to safety checking over non-distributive abstract domains, which we call *Abstract Conflict Driven Learning for Safety*. Chapter 7 present a practical instantiation of ACDLS over the template polyhedra abstract domain for bounded property verification of software netlist designs generated from the hardware RTL circuits.

4.9 Conclusions

Demand for more scalable verification tools is ever growing. In this chapter, we present an alternative solution for verifying hardware, at the heart of which is a verifier for software. To this end, we first translate a design in Verilog RTL into a software netlist in C. We then use state-of-the-art software analyzers to verify the software netlist design. Our experimental results present a comparison of various unbounded verification algorithms at bit-level and software level.

From our experiments, we observe that bit-level operations are less prevalant in conventional software and hence conventional software analyzers are not optimized to reason about the bit-manipulating software netlist design. However, abstract interpretation with manual guidance using Astrée shows superior performance over SAT-based software analyzers. On the other hand, the performance of Astrée is comparable to bit-level hardware model checkers such as ABC for most of our benchmarks. We conclude the chapter by describing the various ways to automatically refine the precision of the analysis in Astrée.

Chapter 5

Monolithic versus Path-based Symbolic Execution of Hardware RTL

In Chapter 4, we presented a novel verification framework for unbounded property verification of hardware RTL designs in software language using native software analyzers. In this chapter, we present a bounded safety verification of hardware RTL via translation to software netlist using two different approaches – *monolithic* and *path-based*.

5.1 Introduction

Bounded model checking (BMC) [22] is one of the major successes of practical formal hardware verification [64] but still runs into crippling scalability limits. Conventional SAT/SMT-based BMC often produces a large formula that is intractable to solve. An alternative, which we investigate in this chapter, is to explore the design one control-flow path at a time, in the manner of KLEE [44].

In this chapter, we address the issue of an explosion in formula complexity in BMC by borrowing the idea of path-based symbolic execution from the domain of software testing and adapting it to hardware verification. In a conventional hardware verification flow, a circuit is synthesized into a netlist for downstream analysis. But it is not straightforward to apply path-based symbolic execution to this representation. Going via translation into a software netlist design directly exposes software execution paths for individual exploration by symbolic execution.

Path-based symbolic execution trades formula size for time complexity but suffers from the potential explosion in the number of paths explored. We address this with a path-based approach in which the verification goal is first decomposed into subproblems, by input-space and state-space decomposition. Each subproblem is then subject to constraints that arise out of the decomposition, which we can exploit during exhaustive exploration of the control paths

in the sub-problem. We embody these techniques in a path-based symbolic execution tool, VERIFOX, that performs automated property-driven slicing and infeasible path pruning to scale formal verification to difficult arithmetic circuits and complex SoC designs. VERIFOX generates path constraints that are easy to solve for two reasons: it generates constraints over a single path at a time, and path fragments are encoded incrementally. The main focus of the technique is to pass only those path conditions to the underlying solver that are feasible with respect to the property under consideration and the given partitioning constraints. These partitioning constraints can be user-specified assumptions or firmware executions that exercise the hardware transition system in a SoC.

Our technique is most suited for circuits that provides opportunity for input-space or state-space decomposition. For example, in a typical SoC design, the firmware only exercise part of the hardware state-space which renders significant numbers of data and control paths infeasible. Conventional BMC tools typically unwind the whole transition system comprising the firmware and hardware components up to a given bound which is then solved using the underlying SAT/SMT solvers. Hence, BMC tools cannot exploit this infeasibility. It may be possible to exploit this infeasibility by applying BMC with CEGAR but we did not explore this direction in this dissertation. On the other hand, techniques such as path-based symbolic execution can identify the infeasibility of the hardware state-space for the given firmware execution during the symbolic execution phase. Hence, this leads to early pruning of infeasible state-space, therefore passing only feasible constraints to the underlying solver. We experimentally demonstrate that this kind of infeasibility can be determined easily for hardware/firmware co-designs that exhibits producer-consumer relationship. In this chapter, we assess this strategy by experimental comparison of VERIFOX, and other path-based symbolic execution tools, to monolithic bounded model checking for verification of the floating-point arithmetic circuits from ARM, a Dual-path floating-point adder/subtractor circuit, and few control-dominated circuits, such as a UART design and a SoC design drawn from open source projects. Our experimental result shows an order of magnitude speedup for some of these large designs consisting of thousands of lines of Verilog RTL.

Claim of Novelty

In this chapter, we make the following novel contributions:

1. We present a path-based symbolic execution tool, VERIFOX, for bounded property verification of software netlist design generated from RTL.

2. We experimentally compare the performance of SAT/SMT-based BMC and path-based symbolic execution for property verification of floating-point arithmetic circuits

from ARM, a dual-path floating-point adder/subtractor circuit, a UART design and a SoC design. Our experimental results show that path-based symbolic execution with optimizations such as aggressive path-pruning strategy and incremental solving outperforms monolithic BMC techniques for the verification of the UART and SoC designs and some arithmetic circuits.

Plan of the chapter

The rest of the chapter is organized as follows. Section 5.2 describes the related work. Section 5.3 presents a motivating example. Section 5.4 describes the bounded verification techniques. Section 5.5 presents the VERIFOX tool flow. Section 5.6 presents the monolithic verification flow for RTL designs. The experimental results are presented in Section 5.7. Section 5.8 concludes the chapter.

5.2 Related work

Symbolic simulation was first used at the gate-level for formal verification [40, 132]. Symbolic simulation [40, 132, 140] of hardware RTL has been used for both verification and testing. The works of [57, 214] perform static analysis of RTL for verification. The authors of [99] combine dynamic simulation with static analysis at the gate level for formal verification.

Traditional approaches for SAT/SMT-based bounded model checking [22, 53, 142] of hardware produce a monolithic SAT/SMT instance which is extremely complex and therefore may be difficult to solve by any state-of-the-art solvers. Input-based case-splitting is a common practice in the semiconductor industry to scale up formal verification [98, 141, 220]. The technique of adding user constraints to cut down the search space has been extensively used in the past, for example, in the IBM RuleBase tool [1]. However, case splitting at the level of SoC has so far been applied in an ad-hoc manner.

The idea of separately verifying different control paths has a long history, for example in Symbolic Trajectory Execution [41]. The work of [153] in STAR and HYBRO combines static as well as dynamic program analysis techniques for test generation in RTL. This work treats RTL hardware as a software program and analyzes the CFG of the RTL design for input vector generation and achieve branch coverage. However, this technique is limited to property verification of restricted shapes. Also, the method is not applicable for property verification of hardware in the presence of a firmware. Compared to our technique where we automatically generate a software representation in C from Verilog RTL, the authors of [153]

[1] https://www.research.ibm.com/haifa/projects/verification/Formal_Methods-Home/

perform manual code instrumentation of the RTL code to symbolically execute the annotated CFG representation of RTL design for test generation. Similar to our technique, this work analyzes the feasibility of every path in the RTL program using a hybrid of concrete and symbolic execution and property based pruning.

The use of high-level structures as a design description for model checking is the holy grail of hardware verification [46, 48, 129, 138, 171, 206]. This is exemplified by word-level hardware verification [143, 206] tools such as word-level STE [46], BMC [24, 171], predicate abstraction [129], interpolation [145] and word-level PDR [52, 218]. Most of these verification tools at the word-level or term-level develop some intermediate tool specific language to represent the hardware RTL.

The concept of symbolic execution [44, 63, 109] is prevalent in the software domain for automated test generation as well as bug finding. This technique is different from the symbolic simulation [132, 140] techniques that are used in the hardware domain. Tools like Dart [109], Klee [44], EXE [45], ExpliSAT [2] and the Cloud9 Framework [150] employ such technique for efficient test case generation and bug finding.

In hardware verification, slicing is performed by *redundancy removal* [127]. For programs, the work of [199] combines code instrumentation, slicing and symbolic execution in the tool *Symbiotic* for detecting locking errors in Linux kernel code. *Symbiotic* uses *klee* for symbolic execution.

5.3 Motivating Example

Figure 5.1 and Figure 5.2 demonstrates the various phases of VERIFOX through a working example. Let us consider a snapshot of the software netlist design generated from the RTL in column 1 of Figure 5.1. The software netlist implements a high-level power management strategy to orchestrate various hardware modules, such as, *core*, *memory* etc. Depending on the interrupt status (env), power modes (mode) and power-gated logic (power_gated), the call to *core* or *memory* is made. These hardware units are complex implementations of a processor core or a memory unit. Column 2 presents the result of property-driven slicing on the input software netlist. This step is purely syntactic, meaning that we perform a backward dependency analysis [217] starting from the property which only preserve those fragments that are relevant to the given property. Similar automatic slicing for VHDL language is proposed by Clarke et al. [57]. Given the property, `(!(reset!=0) || (feedback>0))`, assume that the module memory does not modify `feedback`. Thus, VERIFOX prunes the call to the module memory since this does not impact the property. Given an user-specified

[2] http://researcher.watson.ibm.com/researcher/view_group.php?id=5698

Software netlist (in C)	Sliced software netlist (wrt. property)
`#define threshold 15` `if(reset) {` `mode=TURN_OFF;` `feedback=0;` `} else {` `// Trigger if env is set` `if(env) {` `// check the voltage level` `if(voltage_level < threshold)` `power_gated = 1;` `else power_gated = 0;` `// check the low-power modes` `if(mode == STAND_BY ||` `mode == TURN_OFF) {` `// power gated logic,` `// call to core` `if(power_gated) {` `core(reset,mode,power_gated` `ser_in,&buf_out);` `feedback = LOW;` `}` `else { // normal logic` `core(reset,mode,power_gated` `ser_in,&buf_out);` `feedback = buf_out; }} }}` `else {` `// call to memory` `memory(reset,size);` `}`	`#define threshold 15` `if(reset) {` `mode=TURN_OFF;` `feedback=0; }` `else {` `// Trigger IP if env is set` `if(env) {` `// check the voltage level` `if(voltage_level < threshold)` `power_gated = 1;` `else power_gated = 0;` `// check the low-power modes` `if(mode == STAND_BY ||` `mode == TURN_OFF) {` `// power gated logic,` `// call to core` `if(power_gated) {` `core(reset,mode,power_gated` `ser_in,&buf_out);` `feedback = LOW;` `}` `else { // normal logic` `core(reset,mode,power_gated` `ser_in,&buf_out);` `feedback = buf_out;` `}` `}` `}` `Assertion: (!(reset!=0) || (feedback>0))`

Figure 5.1: Working example demonstrating automated slicing in VERIFOX

assumption, `((reset == 0) && (env == 1) && (mode == STAND_BY) && (voltage_level == 10))`, the result of infeasible path pruning based on the assumption is shown in Figure 5.2. Since `(voltage_level < threshold)`, so the value of `power_gated = 1`, which prunes the call to the core module in the `else` branch. VERIFOX determines the feasibility of paths in the sliced software netlist with respect to the user-provided assumptions using a satisfiability query. An important point to note here is that the number of path constraints after slicing and infeasible path pruning are significantly less compared to the number of paths in the original software netlist design. Additionally, these per-path constraints are much easier to solve compared to a monolithic formula generated from a BMC-style symbolic execution tools.

5.4 Monolithic and Path-based Symbolic Execution

In this section, we describe monolithic and path-based symbolic execution approaches in hardware and software domains. *Symbolic simulation* of hardware designs aims to replace multiple simulation runs, each with different inputs, by a single symbolic simulation run that uses symbolic values as its inputs. The outcome is a set of symbolic expressions for the

Pruned software netlist (wrt. assumptions)

```
Assumption:(reset == 0) &&  (env == 1) &&
(mode == STAND_BY) && (voltage_level == 10)
#define threshold 15
// non-reset logic
else {
if(env) {
 // check the voltage level
 if(voltage_level < threshold)
  power_gated = 1;

 // check the low-power modes
 if(mode == STAND_BY ||
   mode == TURN_OFF) {
  // power gated logic,
  // call to core
  if(power_gated) {
   core(reset,mode,power_gated
   ser_in,&buf_out);
   feedback = LOW;
   }
  }
 }
}
Assertion: (!(reset!=0) || (feedback>0))
```

Figure 5.2: Working example demonstrating path pruning in VERIFOX

outputs of the design. The symbolic expressions are then passed to an appropriate symbolic reasoning engine together with the desired properties.

Symbolic simulation of Verilog RTL

Consider the Verilog RTL example in the left-hand column of Figure 5.3. The circuit has two state-holding registers, each of four bits. Thus, two next-state functions are generated, denoted by x' and y'. The branching in the input Verilog program yields expressions with the *ite* operator, which is shown in the right of Figure 5.3.

Symbolic execution of Software netlist

Column 1 of Figure 5.4 gives the partial snapshot of the software netlist design generated from the Verilog RTL of Figure 5.3. Column 2, 3, 4, 5 of Figure 5.4 gives the path constraints generated by path-based and BMC based tools. The path constraints corresponding to three paths in the program is shown in column [2–4]. Note that all paths in this program are feasible. The monolithic path constraint generated by BMC is shown in column 5. BMC-based tools *always* merges—generating only a *single* bit-vector formula C for a given unwinding bound k. Note that this formula is linear in the size of the program and linear in k even if there is an exponential number of paths in the program. For the constraint in column 5, the symbolic values of variables are computed as expressions over the initial

Verilog RTL	Word-level Simulation
module top(reset,a,b,x,y); **output reg**[3:0] x,y; **input** [3:0] a,b; **input** reset; **always** @(a **or** b) **begin** **if**(reset) **begin** x = 3'b0; y = 3'b0; **end** **else begin** **if**(a > b) x = a+b; **else** y = (a & 3) << b; **end** **end** **endmodule**	x'=*ite*(*reset*,0, *ite*($(a > b)$,$a+b$,x)) y'=*ite*(*reset*,0, *ite*($(a > b)$,y,$(a\&3) \ll b$))

Figure 5.3: Word-level symbolic simulation of Verilog RTL

Software netlist	Path Constraint 1	Path Constraint 2	Path Constraint 3	Monolithic Constraint
void top(){ **if**(reset) { x=0; y=0; } **else** { **if**(a > b) x=a+b; **else** y=(a & 3)<<b; } }	$C_1 \equiv$ $reset_1 \neq 0 \wedge$ $x_2 = 0 \wedge$ $y_2 = 0$	$C_2 \equiv$ $reset_1 = 0 \wedge$ $b_1 \not\geq a_1 \wedge$ $x_3 = a_1 + b_1$	$C_3 \equiv$ $reset_1 = 0 \wedge$ $b_1 \geq a_1 \wedge$ $y_3 = (a_1\&3) \ll b_1$	$C \iff ((guard_1 = \neg(reset_1 = 0)) \wedge$ $(x_2 = 0) \wedge (y_2 = 0) \wedge$ $(x_3 = x_1) \wedge (y_3 = y_1) \wedge$ $(guard_2 = \neg(b_1 >= a_1)) \wedge$ $(x_4 = a_1 + b_1) \wedge (x_5 = x_3) \wedge$ $(y_4 = (a_1\&3) \ll b_1) \wedge$ $(x_6 = ite(guard_2, x_4, x_5)) \wedge$ $(y_5 = ite(guard_2, y_3, y_4)) \wedge$ $(x_7 = ite(guard_1, 0, x_6)) \wedge$ $(y_6 = ite(guard_1, 0, y_5)))$

Figure 5.4: Path-based and monolithic symbolic execution

values of variables m and t, whereas the branching in the program yields expressions with the *ite* operator.

5.5 VERIFOX: Path-based Symbolic Execution of Software Netlist Designs

Figure 5.5 shows the overall tool flow for VERIFOX. Given a hardware design in Verilog RTL and the set of properties in System Verilog Assertion language (SVA), VERIFOX uses the tool V2C to automatically generate the software netlist design in C. VERIFOX can be used for property verification of the software netlist designs. It can also be used for software netlist verification in the presence of a firmware. In this chapter, we show both use cases. Given some user-specified assumptions which specify various partition constraints for state-space decomposition, VERIFOX generates a unified verification model in C from the software netlist design along with these assumptions. In the presence of the firmware,

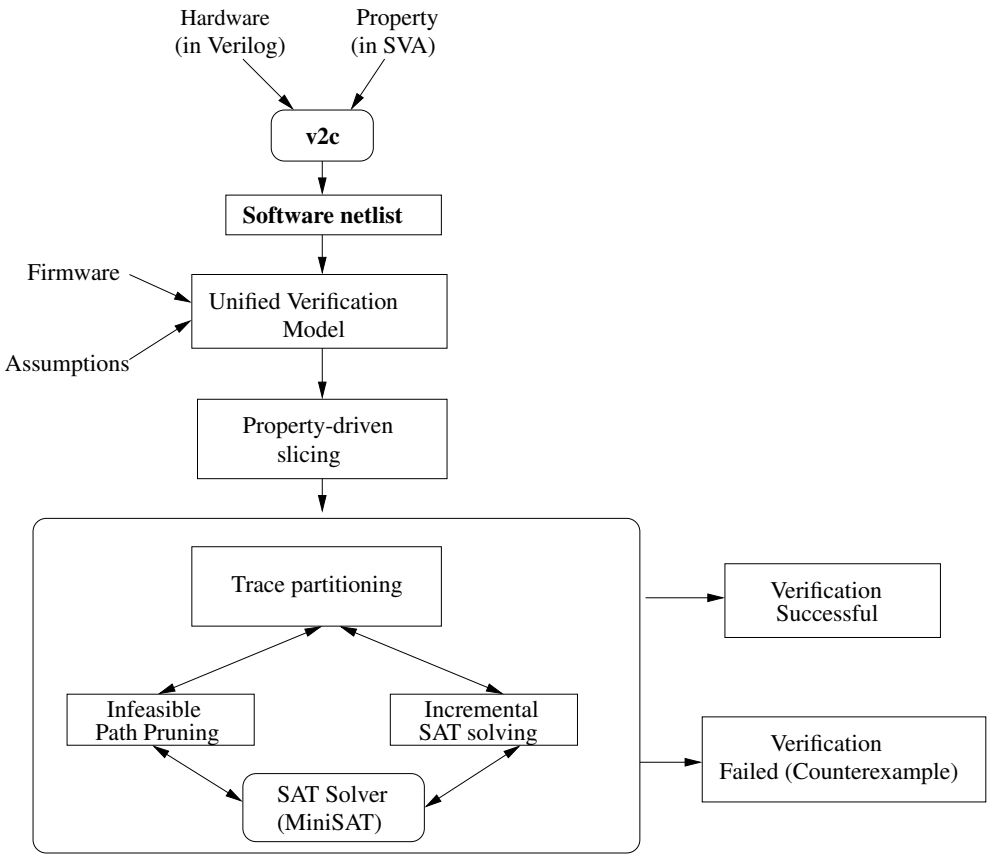


Figure 5.5: VERIFOX tool flow

VERIFOX generates an unified verification model in C from the software netlist design and the firmware, which is explained in Section 5.7.3. We will refer to the unified verification models as software netlist design. The tool then performs property-driven slicing of the software netlist design. The sliced netlist design is then passed to the symbolic execution engine. This engine explores the input sliced netlist design in a depth-first manner with aggressive path pruning strategy. The generated path-constraints are encoded incrementally and passed to the underlying solver only if it is feasible. VERIFOX either shows that the software netlist is correct with respect to the given properties or returns a counterexample trace violating the property.

5.5.1 Working of VERIFOX

VERIFOX is a path-based symbolic execution tool used for bounded safety property verification of Verilog RTL circuits expressed as a software netlist in C language. A typical path-based symbolic execution engine might explore a path until it comes to an $assert(c)$ statement. This whole path can then be posed as a query to a SAT solver to see if the

assertion is violated at that point. If the path is infeasible, the assertion holds trivially. If a large number of paths are infeasible, the symbolic execution may waste time exploring them. VERIFOX employs an eager infeasibility check to prune infeasible paths early, as well as incremental encoding that makes it easier for the underlying SAT solver.

Algorithm 2: Working of VERIFOX

```
   input  : Program P with properties specified with assert(c) statements
   output : The status (Safe or Unsafe) and a counterexample if Unsafe
   /* The initial state                                              */
1  S_0 ← I(x)
2  stmt ← first statement
3  worklist.put((S_0, stmt))
4  while not worklist.empty() do
       /* Depth-first search exploration                             */
5      (S, stmt) ← worklist.get()
6      if stmt is an assume(c) then
7          stmt ← statement after assume(c)
8          if isFeasible(S ∧ c) then worklist.put((S ∧ c, stmt))
9      else if stmt is a branch with condition c then
10         stmt_f ← first statement after stmt if branch is not taken
11         stmt_t ← first statement after stmt if branch is taken
12         if isFeasible(S ∧ c) then worklist.put((S ∧ c, stmt_t))
13         if isFeasible(S ∧ ¬c) then worklist.put((S ∧ ¬c, stmt_f))
14     else if stmt is an assert(c) then
15         stmt ← statement after assert(c)
16         if isFeasible(S ∧ ¬c) then
17             print Unsafe
18             return Counterexample
19         end
20         else worklist.put((S ∧ c, stmt))
21     else
22         S ← Symex(S, stmt)
23         stmt ← the next statement in control flow after stmt
24         if stmt ≠ ⊥ then worklist.put((S, stmt))
25
26     end
27     return Safe
28 end
```

VERIFOX takes a software netlist as an input, where the properties of interest are specified using $assert(c)$ statements in the code. Here, c is a condition stated in terms of program variables that we want to hold at the point where the assertion occurs. If the

condition does not hold, the program is said to have violated the property.

Algorithm 2 shows the overall algorithm of VERIFOX. States mentioned in the algorithm are all symbolic states, which are quantifier-free predicates characterizing a set of program states. Symbolic execution starts with an initial symbolic state $I(x)$, is a quantifier-free predicate over program variables x, and the first statement *stmt* to be executed. Note that we assume all program variables have finite bit-width and thus can be represented as bit-vectors. Every statement acts as a state transformer during the symbolic execution. *worklist* maintains the set of symbolic states, along with the corresponding *stmt* that should execute next.

Assumptions can be specified in the program using $assume(c)$ statements, where c is the condition expressed in terms of program variables. Assumptions restrict the search to only those states for which the condition c holds at the program point where $assume(c)$ is encountered. For example, suppose $(x! = 0)$ characterizes the set of states that have been discovered to be reachable so far by a verification tool. Here, x is a program variable. Upon encountering $assume(x > 0)$, the set of states reachable at the point of assumption is shrunk to only those that satisfy $(x > 0)$. A user can specify assumptions to restrict the verification to only certain regions of the program's state space. (See example in Figure 5.1).

VERIFOX performs a feasibility check at an *assume* statement or a branch (Line 6 and 9) to ensure that only feasible symbolic states are kept in the *worklist*. This ensures that the infeasibility is detected as early as possible. If an assertion is violated, then a counterexample is detected and Algorithm 2 terminates (Line 14). In all other cases, $Symex(S, stmt)$ performs one step of symbolic execution by executing *stmt* from the symbolic state S (Line 22). If no further statement remains to be executed along the path that is being explored, then *stmt* is assigned the value $\perp$. The symbolic state is put in the worklist only if there are further statements remaining (Line 24).

The feasibility checks shown as *isFeasible* pose a query to the underlying SAT solver. Note that Algorithm 2 does not refer to how the methods *worklist.put* and *worklist.get* work. In principle, one can use any search heuristic to select which symbolic state to explore further from the *worklist*. In the current version, VERIFOX employs a Depth-first Search (DFS) strategy of exploration.

Apart from the eager infeasibility check, another crucial optimization is the use of incremental SAT solving during the symbolic execution. VERIFOX can use incremental SAT-solving capabilities in two ways.

In the partial incremental mode, only one solver instance is maintained while going down a single path. Thus, when making a feasibility check from one branch b_1 to another branch b_2 along a single path, only the program segment from b_1 to b_2 is encoded as a constraint

and added to the existing solver instance. Internal solver states and the information that it gathers during the search remains valid as long as all the queries that are posed to the solver in succession are monotonically stronger. If the solver solves a formula ϕ, posing $\phi \wedge \psi$ as a query to the same solver instance allows one to reuse the solver knowledge that it has acquired in the previous query, because any assignment that falsifies ϕ also falsifies $\phi \wedge \psi$. Thus, the solver need not revisit the assignments that it has already ruled out. This results in speeding up the process of feasibility check of the symbolic state at b_2 as the feasibility check at b_1 was *true*. A new solver instance is used to explore a different path after the current path is detected as infeasible.

In the full incremental mode, only one solver instance is maintained throughout the whole symbolic execution. Let $\phi_{b_1 b_2}$ denote the encoding of the path fragment from b_1 to b_2. It is added in the solver as $(B_{b_1 b_2} \implies \phi_{b_1 b_2})$. Then, $B_{b_1 b_2}$ is added as a *blocking variable*, which is also usually known as *solver assumption* [3] to enforce constraints specified by $\phi_{b_1 b_2}$. Blocking variables are treated specially inside the solvers. Unlike regular variables or clauses, the blocking can be removed in subsequent queries without invalidating the solver instance. When one wants to back-track the symbolic execution, the blocking $B_{b_1 b_2}$ is removed and a unit clause $\neg B_{b_1 b_2}$ is added to the solver, thus effectively removing $\phi_{b_1 b_2}$.

Eager infeasibility check may save the engine from exploring multiple paths that emerge through branches encountered after the first point of infeasibility. In our experiments, we find this optimization has a large effect on runtimes. Though VERIFOX poses many queries to the SAT solver, each query is relatively simple due to two reasons: the resultant formula encodes only a single path, and exploration along a path only needs to encode and solve for the path segment (along with the existing constraints) from the last point of query.

5.6 Monolithic Verification using HW-CBMC

HW-CBMC [142] is a formal verification tool for RTL designs in presence of a firmware or a reference model or properties specified as C assertions. Figure 5.6 illustrates the monolithic approach to RTL verification, as implemented in HW-CBMC. Note that the software flow shown in lower part of Figure 5.6 can be used to specify either a firmware or a set of properties of hardware RTL using assertions in C language or a reference model. In our experiments, we show use cases for the first two scenarios. In Section 5.7.2, we use HW-CBMC for RTL verification where the software simply specifies the properties of the RTL in C language. In Section 5.7.3, we use HW-CBMC as a monolithic hardware/software

[3] The SAT community uses the term *assumption variables* or *activation variables* or *assumptions*, but we will use the term blocking variable to avoid ambiguity with assumptions in the program.

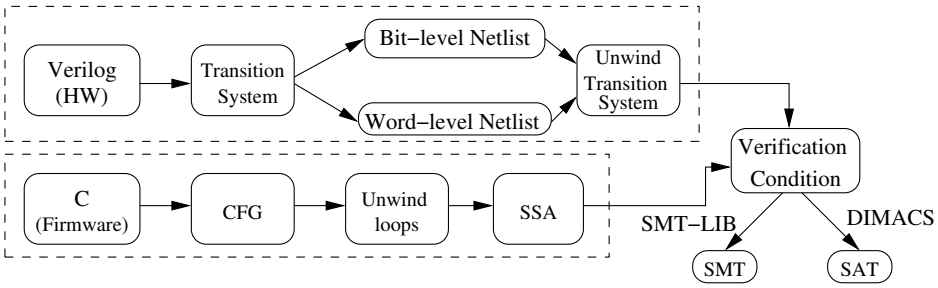


Figure 5.6: Monolithic hardware/software co-verification flow in HW-CBMC

co-verification tool where the software is replaced by a firmware model. In contrast to path-based approach, HW-CBMC maintains two separate flows for hardware and software. The top flow in Figure 5.6 uses synthesis to obtain either a bit-level or a word-level netlist from Verilog RTL. A bit-level netlist is represented in AIG format. A word-level netlist is represented in format that resembles SMT-LIB2 standards. [4] The bottom flow illustrates the translation of C program into static single assignment (SSA) form. Synthesizing netlist from RTL is a standard technique which is widely used in hardware model checkers [63]. Similarly, translating a C program into an SSA representation is a standard technique which is widely used in software verification tools, such as CBMC [54]. HW-CBMC generates a monolithic verification model from AIG and SSA through in-tandem symbolic execution of HW and FW models.

The co-verification model is checked with bounded model checking (BMC). Given an unwind depth k and a co-design property P, BMC operates by unwinding the co-verification model M up to depth k starting from initial state x_0, represented by an initial state predicate I. This results in the following formula which is then checked for satisfiability using an efficient SAT or SMT procedure.

$$I(x_0) \wedge \bigwedge_{i=0}^{k-1} M(x_i, x_{i+1}) \wedge \bigvee_{i=0}^{k} \neg P(x_i)$$

Thus, BMC exploits the finiteness of M and creates k copies of the M for each unrolling. In HW-CBMC, the symbolic execution of hardware and software models are clearly separates and the two flow meet only at the solver phase, where a complex monolithic formula is generated in Conjunctive Normal Form (CNF) from the AIG and SSA. This complex formula is passed on to the solver for the verification purpose.

[4] http://smtlib.cs.uiowa.edu/

5.7 Experimental Results

In this section, we report experimental results for property verification of difficult floating-point arithmetic circuits. We also present verification of an UART and a SoC design in presence of their respective firmware.

5.7.1 Benchmark and Tool Distribution

The source code of VERIFOX is available at `http://www.cprover.org/verifox/`. VERIFOX is developed on top of the CPROVER framework. The website contains the instruction to install VERIFOX. We also distribute the tool binaries for HW-CBMC, CBMC, KLEE and SYMBIOTIC along with the scripts for running these tools in our website. The command line option for running VERIFOX is shown below.

```
verifox-pi <file-name> // partial incremental mode with SAT
verifox-fi <file-name> // full incremental mode with SAT
```

VERIFOX expects the input file to be a C program in `.c` format. The executable `verifox-pi` is configured in partial incremental mode, and the executable `verifox-fi` is configured with full incremental mode.

Benchmarks

A collection of floating-point benchmarks, an UART and a SoC design in Verilog RTL and C along with the firmwares in C is available at `http://www.cprover.org/verifox/`. Due to the proprietary nature of the ARM IP, we cannot distribute the Floating-point arithmetic core from ARM.

Experimental setup

All our experiments were performed on an Intel Xeon machine at 3.07 GHz with 48 GB RAM. MiniSAT-2.2.0 [90] was used as underlying SAT solver with VERIFOX 0.1, CBMC 5.2 [54] and HW-CBMC 5.0 [53]. We compare the performance of VERIFOX 0.1 against other path-based symbolic execution tools, such as KLEE 1.0.0 [44, 109] and SYMBIOTIC 3.0.1 [200]. The timeout for all our experiments is set to 2 hours. All times reported in Table 5.1, Table 5.3, and Table 5.4 are given in seconds.

5.7.2 Property Verification of Floating-point Arithmetic Core

We now discuss the formal verification of a floating-point arithmetic from ARM and verification of a Dual-path floating-point adder.

Floating-point Arithmetic Core from ARM

We verified parts of a floating-point arithmetic unit (FPU) of a next generation ARM GPU. The floating-point core is mainly composed of single and double-precision floating-point Adder, Fused Multiply Add (FMA) and TBL functional units, the register files and other logic to interface with the external environment. The pipelined computation unit implements floating-point operations with operands up to 128-bit. In this chapter, we verified the ADD unit, which performs 128-bit operations by combining two 64-bit adders and produces a result using multiple pipeline stages. Each 64-bit unit can also perform operations with smaller bit widths and is implemented in several thousand lines of complex Verilog, generating several ten thousand gates. The pipelined ADD unit decodes the incoming instruction, applies the input modifiers and provides properly modified input data to the ADD sub-unit. The software implementation of the ARM FPU core is also very complex and consists of several thousand lines of C code.

Dual-path Floating-point Adder/Subtractor

We have developed both a C and a Verilog implementation of an IEEE-754 32-bit single-precision dual-path floating point adder/subtractor, which takes two 32-bit floating point numbers as input and returns a 32-bit floating point number as a result. The floating-point design includes various modules like packing, unpacking, normalizing, rounding and handling of infinite, normal, subnormal, zero and NaN (Not-a-Number) cases. The dual-path adder is inspired from Farmwald's dual-path floating-point addition algorithm [93], which separates the floating-point adder pipeline into two parallel paths that work under different assumptions for optimization purposes. The criterion for partitioning the computation into two paths is based on the exponent difference such that 1) the near path is defined for small exponent differences and 2) the far path is defined for the remaining cases. The dual-path floating-point adder/subtractor is implemented in approximately 700 LOC, generating several hundred gates. The software implementation is approximately 800 LOC in C. The dual-path implementation is substantially different from the ARM FPU.

Reference Model

The floating-point implementation in the CPROVER framework [5] is used as the golden reference model. Note that the CPROVER floating-point reference model [172] is equivalent

[5] http://www.cprover.org/

Design Mode	Bound	Monolithic			Path-wise		
		HW-CBMC		CBMC	VerifOx		
		Bit-level	Word-level	Software	Software		
		Time	Time	Time	Total/Feasible Paths	%-pruning	Time
floating-point adder from ARM							
INF	4	18.12	17.41	**1.01**	7312/12	99.83	16.80 (P)
ZERO	4	18.02	17.24	**1.02**	7312/8	99.89	19.25 (P)
NaN	4	17.87	16.40	**1.01**	7312/8	99.89	14.56 (F)
SUBNORMAL	4	18.73	17.97	**1.47**	7312/3652	50.05	841.25 (F)
NORMAL	4	39.60	34.59	**9.84**	7312/100	98.63	96.47 (F)
Dual-path floating-point adder							
INF	1	0.88	0.87	**0.59**	38648/2	99.99	3.15 (P)
ZERO	1	0.87	0.86	**0.58**	38648/1	99.99	3.11 (P)
NaN	1	0.99	0.86	**0.56**	38648/1	99.99	2.14 (P)
SUBNORMAL	1	169.49	168.36	178.1	38648/5748	85.12	**88.12** (F)
NORMAL	1	22.42	22.38	**22.11**	38648/2971	92.31	55.28 (P)
Mixed	1	667.17	664.42	749.74	38648/38072	1.49	**484.76** (F)
No Partition	1	668.61	645.98	642.50	38648/38648	NA	**497.46** (F)

Table 5.1: Run times for property verification of floating-point arithmetic circuits (All times are in seconds) (NA denotes Not Applicable)

to the Softfloat [6] reference model [170]. The tools HW-CBMC, CBMC 5.2 and VERIFOX use the same built-in golden reference model as implementated in the CPROVER framework.

Discussion on the results

Table 5.1 reports the run times for property verification of pipelined ARM FPU and the dual-path floating-point arithmetic circuit. We verify the functional correctness of the floating-point implementations.

We construct a miter circuit in HW-CBMC to verify the correctness of the floating-point implementations. In general, a miter circuit is built from two given circuits A and B as follows – identical inputs are fed into A and B, and the outputs of A and B are compared using a comparator. We consider the case in which one of the circuits is an implementation and the other is the golden specification. Figure 5.7 shows an example miter for checking combinational equivalence of a 32-bit floating-point adder/subtractor circuit. We provide floating-point numbers `f`, `g` as inputs to the specification model (built inside the tool). The same input is passed to the hardware implementation (in Verilog RTL) using a function set_inputs (). Subsequently, we indicate that we want to perform a floating-point addition by setting isAdd=1. The reference model computes the floating-point addition using the statement float C_result=f+g. The result computed by the hardware design and the reference model are compared using the compareFloat() function. Construction of the miter is similar in CBMC and VERIFOX, except that the implementation model is the

[6] http://www.jhauser.us/arithmetic/SoftFloat.html

```
void miter(float f, float g) {
  // setting up the inputs to hardware FPU
  fp_add_sub.f=*(unsigned *)&f;
  fp_add_sub.g=*(unsigned *)&g;
  fp_add_sub.isAdd=1;
  // propagates inputs of the hardware circuit
  set_inputs();
  // get result from hardware circuit
  float Verilog_result=*(float *)&fp_add_sub.result;
  // get result from specification model
  float C_result=f + g;
  // compare the outputs
  assert(compareFloat(C_result, Verilog_result));
}
```

Figure 5.7: Miter for combinational equivalence checking for a 32-bit floating-point adder/-subtractor for the case of addition

software netlist design of the floating-point circuit and the specification model is the same as in HW-CBMC.

The floating-point design is highly modular and can be partitioned based on different parameters – the type of input numbers, the type of rounding modes, or the exponent difference among others. Column 1 gives the name of partition mode – INF, ZERO, NaN, SUBNORMAL, NORMAL. For example, the partition constraint "INF" means the addition of two infinite numbers. Note that we use two additional partition constraints for the "Mixed" (combination of all types of numbers) and "No Partition" modes in the dual-path adder. Column 2 gives the bound up to which the transition system is unwound. This is equal to 1 for the dual-path adder because it's a combinational design, but the bound has to be increased for the pipelined designs to account for the cycles it takes to produce a valid result. Column 3 and column 4 present the verification times for the hardware implementation of floating-point circuits at bit-level and word level using HW-CBMC. Column 5 and Column 8 give the verification times for the software implementation of floating-point arithmetic circuits using CBMC and VERIFOX respectively. Note that we report the best times for incremental solving using VERIFOX, which is denoted by Partial (P) and Full (F) incremental mode in column 8. Column 6,7 gives the total/feasible path count for each partition constraint and the percentage of pruned paths by VERIFOX. The focus of our experiment is to compare property verification of complex floating-point arithmetic circuits using monolithic and path-based symbolic execution tools.

Software analyzers such as CBMC and VERIFOX are used to verify the software netlist design of the floating-point arithmetic core. On the other hand, HW-CBMC synthesizes the design into a bit-level netlist or word-level netlist. Input-based case splitting is a common practice to scale property verification and equivalence checking for difficult floating-point arithmetic circuits. We use assumptions to partition the traces in the floating-point circuits.

These assumptions are often specified as conjunctions of different constraints where a constraint may be a combination of the type of operation (ADD, SUB etc.), type of rounding-modes (RNE, RTZ etc.), type of input numbers (INF, NaN, Subnormal etc.) and other parameters.

Our experimental results demonstrate that monolithic and path-based symbolic execution techniques exploit these assumptions differently. BMC tools such as HW-CBMC and CBMC generate a monolithic formula that encodes the whole floating-point circuit along with the assumptions which are then passed to the backend solver. The solver uses these assumptions to partition the circuit through constant propagation. On the other hand, VERIFOX performs path-based symbolic execution with eager infeasibility checking, which prunes away all those paths that do not satisfy the assumptions. This guarantees that all the path constraints which are passed to the solver are consistent with the assumptions. We investigate the reason for higher verification times taken by all the tools for adding subnormal and normal numbers compared to infinity, NaNs and zeros. This is attributed to the higher number of paths in subnormal and normal cases compared to INF, NaN's and zero's as given in column 6.

Column 5 in Table 5.1 shows that CBMC wins consistently over HW-CBMC and VERIFOX for the verification of ARM FPU. We investigate the structure of ARM FPU and observe that the pipelined implementation forces VERIFOX to traverse deep in a particular path and then backtrack to a much higher level in the symbolic tree due to the infeasibility of the current path. This causes VERIFOX to throw away several path fragments which were considered feasible while going deep in the path. This results in the wastage of significant computation time resulting in a better relative performance by CBMC as compared to VERIFOX in this case. However, VERIFOX wins with almost 1.5$\times$ speedup over CBMC for the dual-path adder for the addition of subnormals, "Mixed" mode and even without any partitioning ("No partition"). This implies that constraints generated from path-based symbolic execution are easier to solve compared to translating the entire dual-path floating-point arithmetic logic into a monolithic formula. Along with this, the dual-path adder contains a state-machine that implements separate cases for the addition of different types of numbers. This allows VERIFOX to perform early infeasibility check and prune most of the irrelevant logic upfront in the symbolic execution phase using the assumptions. Thus, our results demonstrate two important things – 1) the performance of monolithic and path-based symbolic execution heavily depends on the structure of the floating-point circuit implementation, and 2) path-wise symbolic execution can be used for automatic bounded safety verification of complex arithmetic designs.

5.7.3 Property Verification of RTL in the Presence of Firmware

We now present a second application for monolithic and path-based tools for hardware RTL verification in the presence of firmware (FW) model. The safety properties are specified jointly over the hardware and software interfaces [152]. The verification of a FW given in C/C++ together with an RTL given in Verilog/VHDL against a co-specification model is typically known as *Hardware/Software Co-verification problem* [152, 169, 173]. In this chapter, we use a simplistic use case for hardware/software co-verification to demonstrate the application of monolithic and path-based tools. The UART and the SoC design used for our experiments shows sequential behavior which follows a producer-consumer relationship between the hardware and firmware [173]. Hence, we can sequentialize the interaction between the RTL and its interacting firmware. This enables the application of sequential verification tools to verify these use cases. However, hardware/software co-designs typically exhibit concurrent behavior, for which sequential tools may not be applicable.

Hardware/software co-verification is broadly performed at two different phases in the design cycle: pre-Register Transfer Level (pre-RTL) and post-RTL. In this experiment, we focus on the post-RTL co-verification. In the post-RTL phase, verification is mandatory to ensure that the driver or FW is executed correctly together with the RTL. For monolithic verification, we use HW-CBMC for verifying the Verilog RTL in presence of the firmware, which is specified in C. We are not aware of any other formal hardware/software co-verification tools other than HW-CBMC that support C and Verilog RTL.

For path-based verification, the RTL is translated into the software netlist. Given the firmware and the software netlist design generated from RTL, a single-threaded unified sequential design is generated in C for path-based symbolic execution. We will explain the generation of this unified sequential design with an example. Figure 5.8 shows a fragment of the firmware model for the UART design. This firmware model is used by the tool HW-CBMC and is considered as base line implementation for our experiments. For a pure software implementation using software analyzers, this firmware implementation is translated into a compilable C program by replacing the HW-CBMC related primitives. Note that the functions set_inputs () and next_timeframe () in the firmware of Figure 5.8 are the primitives of HW-CBMC that direct the tool to set the inputs to the hardware signals and advance the clock signal respectively. We replace the next_timeframe () primitive in the firmware by a call to the top-level procedure of the software netlist design of the UART. The calls to set_inputs () are skipped since the values of input ports are automatically passed from the firmware to the software netlist design. This generates a single-threaded unified sequential design in C from the software netlist design and the translated firmware. We then apply various sequential software analyzers for verifying this unified sequential software design.

Circuit	Verilog LOC	Latches(L)/ FF	Input Ports	Output Ports	GATE Count	Firmware (LOC)
Universal Asynchronous Receiver Transmitter						
UART	1200	356	12	9	413	528
System-On-a-Chip						
SoC	3567	840	14	11	945	734

Table 5.2: Design statistics for UART and SoC Design

Property Verification of the UART Design

We now discuss the formal verification of a UART design in the presence of a firmware.

About UART

A UART design is a Universal Asynchronous Transmitter/ Receiver, which is used for the asynchronous transmission and reception of data which provides serial communication capabilities with a modem or other external devices. The UART is compliant with industry standards for UART and interfaces with the wishbone bus. The core supports single transmission/reception format which is 1 start bit, 1 stop bit and 8 data bits without parity. It also contains a single transmit/receive buffer. The UART implements a baud rate generation circuit based on a harmonic frequency synthesizer that supports a wide variety of clock frequencies. The UART is implemented in approximately 1000 lines of Verilog RTL, generating several hundred gates. The UART is obtained from `http://www.opencores.org`. We translate the UART into a software netlist in C using *v2c*. The software netlist is approximately 1100 LOC. The design statistics of UART design are given in Row 1 of Table 5.2.

Firmware Model of UART

Figure 5.8 shows a fragment of the firmware model for the UART design which is used by HW-CBMC. The firmware implements linux style inb() and outb() functions to communicate with the hardware ports. The wb class of functions communicate with the wishbone bus interface. The set_inputs () and the next_timeframe() functions are the primitives of HW-CBMC that direct the tool to set the inputs to the hardware signals and advance the clock signal respectively.

The operation in the firmware begins by resetting the UART which is followed by a wb_idle() function. Then, the actual operation starts by setting the appropriate values to the memory mapped registers which are propagated to the hardware modules. In this example,

Wishbone Interface

```
typedef unsigned char u8;
unsigned char inb
 (unsigned long port) {
  return wb_read(port);
}
void outb (u8 value,
 unsigned long port) {
  wb_write(port, value);
}
void wb_reset(void) {
 rtfSimpleUart.rst_i = 1;
 set_inputs();
 next_timeframe();
 rtfSimpleUart.rst_i = 0;
 rtfSimpleUart.stb_i = 0;
 rtfSimpleUart.cyc_i = 0;
}
void wb_idle() {
 set_inputs();
 next_timeframe();
}
void wb_write(_u32 addr,
              _u8 b) {
 rtfSimpleUart.adr_i = addr;
 rtfSimpleUart.dat_i = b;
 rtfSimpleUart.we_i = 1;
 rtfSimpleUart.cyc_i = 1;
 rtfSimpleUart.stb_i = 1;
 set_inputs();
 next_timeframe();
 rtfSimpleUart.we_i = 0;
 rtfSimpleUart.cyc_i = 0;
 rtfSimpleUart.stb_i = 0;
}
```

Firmware

```
int main() {
wb_reset();
wb_idle();
// Configure the uart
outb (0x13, UART_MC);
outb (0x80, UART_CM3);
outb (0x00, UART_CM2);
outb (0x00, UART_CM1);
outb (0x00, UART_CR);
outb (0x03, UART_IE);
char tx_b[] = "Hello world";
_u8 istatus = 0;
char rx_b[100];
int i=0,c=0,d=0;
for (i=0; i<1990; i++){
if (rtfSimpleUart.irq_o){
 istatus=inb(UART_IS)&0x0c;
 if(istatus==0x0c){
 //it was a tx_empty interrupt
 outb(*(tx_b+c),UART_TR);
 c++;
 }
 else{//istatus==0x04
 //it was an rx_data interrupt
  rx_b[d] = inb(UART_TR);
  d++;
 }
}else {
 // no interrupt.
 wb_idle();
 wb_idle();
 }
}
for(i=0; i<=10; i++)
 assert(rx_b[i] == tx_b[i]);
}
```

Figure 5.8: Firmware model for UART IP

the firmware sets the UART in loopback mode with interrupts enabled. We verify properties which capture the interaction between the hardware and the firmware. In particular, we verify whether the transmitted data is same as the received data in the loopback mode. This is captured by writing an assertion in the firmware model that checks the data from the transmitter and the receiver modules.

UART Configuration

The UART design is configured in different operating modes, namely–*transmission without interrupt enabled* (Mode A), *transmission with interrupts enabled* (Mode B) and *loopback mode with interrupts enabled* (Mode C). The data-width varies in each mode and ranges from 8-bits to 64-bits. For Mode A and Mode B, the firmware exercises only the transmitter module to transmit non-deterministic data through the serial output. The receiver module is inactive in mode A and mode B. For Mode C, the UART is configured in loopback mode with interrupt logic enabled. Thus, both the transmitter and the receiver are active in this

mode. The control is non-deterministic in this mode since the firmware can receive an interrupt from the transmitter or receiver module depending on whether the transmitter buffer is empty or the receiver buffer is full respectively.

Discussion

Table 5.3 reports the run times for bounded safety verification of the UART design. Column 1 in Table 5.3 gives the name of the design mode – transmission without interrupts *(transmit)*, transmission with interrupts enabled *(trans_intr)* and loopback mode *(loopback)*. Column 2 gives the maximum loop bounds in the firmware. Column 3 and 4 report the verification times using HW-CBMC at the bit-level and word-level. Column 5,7 and 10 give the verification times for verifying the software netlist design of the UART using CBMC, KLEE, SYMBIOTIC and VERIFOX respectively. Column 8 and 9 report the total/feasible path counts and the percentage of infeasible paths pruned for each mode. The Timeout entries in Table 5.3 is denoted by TO. The focus of our experiment is to demonstrate the scalability of path-based techniques over monolithic BMC instances for bounded safety verification of RTL in presence of the firmware model. To this end, we compare HW-CBMC and CBMC to other path-based software analyzers.

Safety properties We verify several end-to-end properties of the hardware/firmware interactions of the UART. In Mode A and Mode B, we verify whether the transmitted data (32-bit or 64-bit) is available through the serial output port after a pre-determined number of clock cycles. In both these configurations, the firmware only exercises the transmitter module by appropriately configuring the memory-mapped registers. Hence, VERIFOX can prune the receiver module since this is infeasible for Mode A and Mode B. In Mode C, we verify whether the transmitted data matches the data received when the UART is configured in loopback mode. In this mode, both the transmitter and receiver are active.

Detecting UART bugs We detect 3 bugs in the UART design. All the bugs are manifested in the UART hardware logic. We describe the sources of these bugs below.

- Bug in *transmit* mode – This bug occurs when the transmitted data overlaps with start and stop bits resulting in an incorrect bit pattern in the serial output of the transmitter.
- Bug in *trans_intr* mode – This bug triggers in the control-path of the transmitter module which deactivates the transmitter *empty* signal. This illegally prevents any transmitter interrupt from occuring even when the transmitter buffer is empty, thereby violating the data transfer protocol.

- Bug in *loopback* mode – This bug manifests when the data is present in the receiver buffer, but the *data_present* signal is never asserted, as a result of which the receiver interrupt is never issued in the firmware. Thus, the transmitted data is not the same as the received data in loopback mode.

The result in Table 5.3 clearly shows that path-based symbolic execution is the clear winner over monolithic BMC for all configurations (marked in bold). The average speedup for proving safety properties is on average 21× for VERIFOX over HW-CBMC. VERIFOX performs better than KLEE for most configurations in Mode A and Mode C. In Mode B, KLEE marginally wins over VERIFOX by a fraction of a second. CBMC performs poorly for most configurations. This is partly due to the higher bounds (> 500) of the loops in the firmware. The loop unwinding process in CBMC is slow and takes significantly longer time than unwinding in HW-CBMC.

The bottom part of Table 5.3 report the verification times for detecting various bugs in the UART design. The result shows that VERIFOX is the fastest in detecting both the data-path and control-flow bugs for all configurations. The average speedup for detecting bugs is on average 48 times for VERIFOX over HW-CBMC.

Note that both KLEE and VERIFOX are run with the same configurations – depth-first exploration strategy, eager infeasibility check, and incremental solving. KLEE uses STP theorem prover and VERIFOX uses MiniSAT 2.2.0 solver in the backend.

In a typical hardware/firmware interaction, the firmware exercises only a fragment of the hardware state-space. This renders many interactions between the firmware and hardware infeasible. For example, the *transmit* mode restricts the communication to data transfer between the firmware and the serial output port of the transmitter module in the UART while completely bypassing the receiver module. This enables path-based tools to prune the receiver logic owing to its infeasible behavior. Hence, it significantly reduces the number of paths to be considered while making the generated SAT solver queries much simpler to solve. By contrast, monolithic BMC tools retain the receiver logic in their verification condition and rely on SAT solvers to efficiently solve them.

Property Verification of System-on-Chip

About SoC

We obtained an open source System-on-Chip design from [205]. It consists of an 8051 micro-controller, a memory arbiter, an external memory (XRAM) and cryptographic accelerators, as shown in Figure 5.9. The design statistics of the SoC design are given in Table 5.2. The accelerator implements encryption/decryption using the Advanced Encryption Standard

Mode	Bound Depth	Monolithic			Path-based				
		HW-CBMC		CBMC	Klee	Symbiotic	VerifOx		
		Bit-level Time	Word-level Time	Software Time	Software Time	Software Time	Total/Feasible Paths	%-age Pruning	SAT Time
non-deterministic data but deterministic control (Mode A)									
transmit (32)	250	15.02	15.78	108.25	**0.98**	29.36	247104/224	99.90	1.13
transmit (64)	500	23.87	23.29	206.89	1.68	29.90	247104/324	99.86	**1.61**
non-deterministic data and non-deterministic control (Mode B)									
trans_intr (32)	250	14.86	15.98	69.19	**1.15**	29.53	247104/295	99.88	1.49
trans_intr (64)	500	24.14	26.42	117.73	**1.27**	29.58	247104/362	99.85	1.81
non-deterministic data and non-deterministic control (Mode C)									
loopback (8)	230	52.06	55.71	1289.85	10.76	29.01	247104/354	99.85	**3.95**
loopback (16)	500	122.12	147.65	5410.18	32.24	29.04	247104/690	99.72	**12.89**
loopback (32)	650	170.62	192.35	TO	112.32	28.27	247104/1282	99.48	**21.85**
loopback (64)	1300	409.71	445.66	TO	494.25	56.71	247104/2566	98.96	**62.31**
detecting data-path bugs in transmission mode w/o interrupt									
transmit (64)	520	28.43	28.14	186.99	**0.96**	26.67	247104/324	99.86	1.12
detecting control bugs with interrupt enabled									
transmit (64)	520	31.35	29.70	113.57	1.18	27.13	247104/362	99.85	**1.05**
detecting control bugs in loopback mode									
loopback (64)	1300	443.15	427.68	TO	492.59	54.23	247104/2566	98.96	**62.34**

Table 5.3: Bounded safety verification of UART

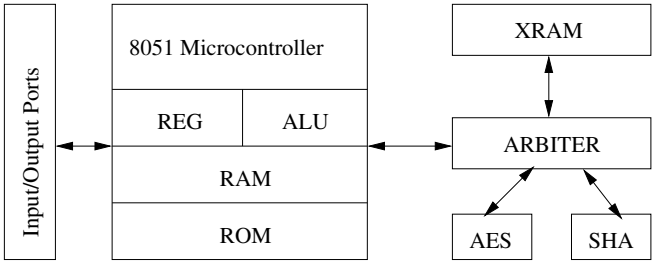


Figure 5.9: SoC Design obtained from [205]

(AES). There is a separate module that interfaces the AES to the 8051 micro-controller using a memory-mapped I/O interface. The microcontroller communicates with the accelerators and the XRAM by reading or writing to XRAM addresses. The arbitration of these modules is done by the memory arbiter module. We translate the whole SoC to software netlist in C. The number of lines of code in the software netlist is approximately 3200 LOC.

About Firmware

The firmware initiates the operation in the SoC by first writing to an initial memory-mapped register. The firmware implements linux-style inb() and outb() function calls which are used to communicate with the input and output ports of the RTL. Once the initialization phase is complete, the cryptographic accelerators use direct memory access to fetch the data from the external memory. The completion of the operation is determined by polling the appropriate memory-mapped register. The XRAM and the accelerator are connected to

Mode	Bound Depth	Monolithic			Path-based			
		HW-CBMC		CBMC	Klee	Symbiotic	VerifOx	
		Bit-level	Word-level	Software	Software	Software	Software	
		Time	Time	Time	Time	Time	%-age Pruned	Time
non-deterministic data and non-deterministic control (safe)								
data_transfer	20	86.92	86.23	104.25	25.98	29.36	68	**17.42**
AES_feedback	30	102.64	110.34	156.82	45.21	52.82	54	**33.89**
non-deterministic data and non-deterministic control (unsafe)								
write_RAM	20	92.63	91.28	95.76	5.68	28.14	92	**1.27**

Table 5.4: Bounded safety verification of SoC

two separate ports of the micro-controller. The arbitration of these modules is done by the memory arbiter module.

Discussion of the result

Table 5.4 reports the runtimes for bounded safety verification of the SoC design. Column 1 in Table 5.4 gives the name of the design mode – transmission with interrupt (data_transfer), communication with AES (AES_feedback), and writing to XRAM (write_XRAM). Column 2 reports the maximum loop bound in the firmware. Column 3 and 4 report the verification times using HW-CBMC at the bit-level and word-level. Columns 5,6,7 and 9 report the verification times for checking the software netlist design using CBMC, KLEE, SYMBIOTIC and VERIFOX respectively. Column 8 gives the percentage of infeasible paths pruned. Note that both full and the partial incremental solving times are comparable in this case. So, we only report the full incremental solving times for VERIFOX.

Proving safety properties of SoC

The firmware writes a sequence of non-deterministic data to the XRAM port and then reads the data from the same port. We verify various safety properties over the interaction between the hardware and the firmware. In particular, we verify whether the sequence of non-deterministic data transmitted through outb() is the same as the sequence of data received through inb(). Furthermore, we verify that reading/writing to the appropriate memory-mapped registers produces the correct result during the data transmission phase.

We detect a bug in the SoC design. This bug manifests itself when the memory arbiter hardware wrongly arbitrates the port selection which force the write strobe for the external RAM to be LOW. This condition violates the data transfer protocol in the SoC design.

The result in Table 5.4 shows that VERIFOX is approximately 5× faster than HW-CBMC and 6× faster than CBMC for proving safety. For detecting bugs, the speedup is 72× and 75× for VERIFOX over HW-CBMC and CBMC. However, VERIFOX marginally wins over

other path-based tools such as SYMBIOTIC and KLEE for safety proofs as well as bug finding. The result shows that the verification times for all path-based techniques are comparable.

The firmware exercises only the micro-controller logic and transfer sequence of bytes to the XRAM port bypassing peripherals connected to other ports such as hardware accelerator. This configuration allows path-based tools to prune the logic for accelerator which otherwise could not have been removed by simple property-driven slicing.

Conventional monolithic hardware verification tools such as HW-CBMC synthesize the hardware and the software separately and generate constraints from each of them in two separate flows. These two flows meet only at the level of the solver. Thus, HW-CBMC is not able to automatically prune the hardware state-space with respect to the firmware executions, but relies on SAT solvers to prune the irrelevant logic. Hence, these tools pass extremely complex formulas to the underlying solver which is difficult to solve by MiniSAT solver used for our experiments. By contrast, VERIFOX automatically determines the infeasible path constraints originating from the interaction between the hardware and firmware during the symbolic execution phase and prunes them straightaway, hence it does not pass irrelevant path constraints to the underlying solver. It is important to note that forward symbolic execution without the optimizations such as path-pruning and slicing, timeout for all the benchmarks.

5.8 Conclusions

In this chapter, we present a path-based symbolic execution tool, VERIFOX, for bounded safety verification of software netlist designs generated from the hardware RTL. We have shown that depending upon the structure of the RTL design, it can be beneficial to apply path-based symbolic execution techniques as opposed to monolithic BMC-style verification. Optimizations such as eager path pruning combined with incremental solving enables VERIFOX to be competitive with other tools. Experimental results show an order of magnitude speedup for property verification of the UART and the SoC design compared to the competing monolithic and path-based tools. For the property verification of the floating-point arithmetic designs, VERIFOX shows better performance on the dual-path floating-point adder/subtractor design. By contrast, monolithic BMC-style verification using CBMC performs better than VERIFOX on the floating-point arithmetic core from ARM.

Chapter 6

Generalizing CDCL to Safety Checking over Relational Domains

This chapter presents a new abstract interpretation framework for generalizing the CDCL architecture to bounded safety verification over template-based abstract domains. To this end, we show that the CDCL algorithm employed in modern day satisfiability solvers can be used to compute fixed points over lattices of program traces to determine safety. Given an unsafe trace transformer that finds the set of valid and unsafe traces and a safe trace transformer that finds the set of invalid or safe traces, satisfiability or model finding can be seen as a property of fixed points of trace transformers over this lattice. The CDCL algorithm alternates between an overapproximate model search phase and an underapproximate conflict analysis phase. We show that the model search computes a greatest fixed point of unsafe trace transformers using deductions and decisions. The conflict analysis computes a least fixed point of safe trace transformers over a downset lattice with a heuristic choice of conflict reasons. Thus, the CDCL architecture can be seen as a natural tool to build a sound and precise safety verifier that uses decision and learning to automatically refine the precision of an analysis. The goal of this new theoretical framework is to perform precise abstract interpretation of the software netlist designs generated from the RTL using CDCL-style transformer refinement over the non-distributive lattices. In this chapter, we refer to software netlist as program.

6.1 Static Analysis and Bounded Model Checking

Static program analysis with abstract interpretation [69] is widely used to verify properties of safety-critical systems. Static analyses commonly aim at computing program invariants as fixed-points of abstract transformers. Abstract states are chosen from a lattice that has meet ($\sqcap$) and join ($\sqcup$) operations; the meet precisely models set intersection (or conjunction,

taking a logical view), and the join overapproximates set union (or disjunction). Over-approximation in the join operation is one of the sources of precision loss, which can cause false alarms. Typical abstract domains are *non-distributive*; In non-distributive abstract domains, analyzing program behaviours separately can improve the precision of the analysis. Usual means to address false alarms therefore include not only the use of richer abstract domains, but also of refinements that delay joins or perform some form of case-splitting. Such techniques trade off higher precision against lower efficiency and may be susceptible to case enumeration behaviour.

By contrast, Model Checking (MC) [9] can be seen to operate on distributive lattice structures that represent disjunction without loss of precision. Classical MC directly operates on distributive representations, such as BDDs, while more recent implementations use SAT solvers. SAT solvers themselves operate on partial assignments, which are non-distributive structures. To handle disjunction, case-splitting is performed [89]. Propositional SAT solvers can reason about large and complex formulae, and are often able to avoid enumerating cases. The impressive performance of CDCL solvers is credited to well-tuned decision heuristics and sophisticated conflict-driven learning algorithms. An appealing idea is to lift CDCL from the domain of partial assignments to other non-distributive domains.

Abstract Conflict Driven Clause Learning (ACDCL) [86, 87, 122] is one such lattice-based generalization of CDCL. ACDCL is a general algorithmic framework, parameterized by a concrete domain C and an abstract domain A. Classical CDCL can be viewed as an instance of ACDCL in which C is the set of propositional truth assignments and A the domain of propositional partial assignments [122]. Since the concrete domain is a parameter to the framework, ACDCL can in principle be used to build both *logical decision procedures* [32] and *program analyzers* [88, 174]. In the former case, the concrete domain is the set of candidate models for the formula; in the latter case, it is the set of program traces that may lead to an error.

In this chapter, we present an abstract interpretation framework for generalizing CDCL to precise safety verification over the relational as well as non-relational abstract domains. We call our framework *Abstract Conflict Driven Learning for Safety* (ACDLS). The key insight of ACDLS is to use decisions and learning to precisely reason about disjunctions in non-distributive domains, thereby automatically refining the precision of analysis for safety checking of programs. We introduce two central components of our framework: an abstract model search algorithm that uses decisions and propagations to search for the counterexample trace and an abstract conflict analysis procedure that performs automatic transformer refinement using learning.

6.1.1 Connection between CDCL and Program Analysis

A key insight that connects CDCL to program analysis is that they use an imprecise over-approximate domain of partial assignments to gain efficiency and techniques like decision and clause learning to improve precision. Silva et al. in [87] propose an understanding of CDCL in the language of lattices and transformers suggesting a "*Grand Unification*" of SAT and static analysis. In this dissertation, we take one step towards this unification goal by presenting an abstract interpretation framework for generalizing CDCL to a fixed point computation procedure over a lattice of program traces to determine program safety. A practical benefit of this generalization is that it gives an architectural framework for developing abstract interpretation tool that can *efficiently* and *precisely* reason about disjunctive properties over the non-distributive abstract domains. One such practical instantiation of ACDLS [174] is presented in Chapter 7.

Claim of Novelty

In this chapter, we make the following novel contributions.

1. We present an abstraction of trace semantics that is suitable for lifting the CDCL architecture to non-distributive abstract domains.
2. We characterize satisfiability by fixed point computation of the trace transformers.
3. We present an abstract model search algorithm that computes a greatest fixed point of the unsafe trace transformer on an overapproximating domain using deductions and decisions.
4. An abstract conflict analysis algorithm is used to compute a least fixed point of safe trace transformer over a downset lattice with heuristic choice of conflict reasons.
5. We present a soundness, completeness and termination proof of the ACDLS algorithm.

Plan of the chapter

The rest of the chapter is organized as follows. The related work is described in Section 6.2. Section 6.3 gives the definitions. Section 6.4 gives the program model. The semantic representation of the program is described in Section 6.5. Section 6.6 describes CFG safety and defines various transformers over traces, with approximations of trace semantics given in Section 6.7. Section 6.8 gives the lattice of approximation of trace semantics which includes the control flow lattice and lattice over static single assignments. The abstract conflict driven learning for safety framework is explained in Section 6.9, along with the

soundness, completeness and termination proofs in Section 6.10. Section 6.11 concludes the chapter.

6.2 Related Work

In this section, we discuss previous works which use satisfiability procedures for static analysis or build satisfiability procedures based on abstract domains and work that attempts to unify the treatment of satisfiability procedures and program correctness in the same framework.

Use of satisfiability procedures in abstract interpretation

Static program analysis using abstract interpretation [69] is widely used to verify properties of safety-critical systems. It analyzes the program over an abstract domain which contains an approximate representation of the program behavior. Imprecision in the static analysis may be due to the use of a less expressive abstract domain such that it can not reflect the properties of the program, or due to merge operations or use of overapproximate abstract transformers or imprecise fixed point iteration through widenings.

Precision loss due to merging can be handled through the systematic exploration of program paths, as in trace partitioning [182], or through the application of CDCL-style reasoning [88, 174], DPLL(T) [124], or through unification [210]. The works of [181] and [209] synthesize best abstract transformer using satisfiability solvers to handle imprecision from overapproximate transformers. On the other hand, the precision loss due to fixed point iteration is addressed by Monniaux et al. in [166], by leveraging SMT solvers to improve the precision of invariant generation. This technique performs abstract fixed point iterations by focusing on certain paths in the program and choosing subsequent paths via bounded model checking using SMT solvers. Alternatively, strategy iteration techniques [102] are used to compute precise fixed points over infinite height abstract domains without using widening. The works of [25, 26, 151] present purely logic based account of program verification. The work of [88] presents a CDCL-style program analyzer that uses decision and learning over Interval domains. A similar technique that lifts DPLL(T) to programs is Satisfiability Modulo Path Programs (SMPP) [124]. SMPP enumerates program paths using a SAT formula, which are then verified using abstract interpretation. The work of [165] proposes an algorithm inspired by constraint solvers for inferring disjunctive invariants using intervals. [32] use decision and learning for heap-manipulating programs, but uses a simplistic conflict analysis. In this dissertation, we generalize CDCL to safety checking over a template-based abstract domain using the framework of abstract interpretation.

Developing satisfiability procedures based on abstract domains

The lifting of CDCL to first-order theories is proposed in [65, 82, 160]. The DPLL(T) framework provides an unified approach to developing decision procedures [100]. However, the separation between the Boolean and theory solver in DPLL(T) can lead to performance issues. Hence, several frameworks evolved based on DPLL(T) in the past. Some of the attempts in this direction are Abstract DPLL framework [177], generalized DPLL [160], natural domain SMT [65]. The work of [32] developed a floating-point decision procedure that lifts CDCL to Interval domains, whereas Nelson-Oppen combination procedure [176] was lifted to abstract domains [209] and [120] lifted Stålmarck's method to arithmetic logic. Fränzle et al. [97] present a tight integration of SAT solving with interval-based constraint solving to handle large constraint systems. [179] describes a constraint solving algorithm using relational numerical domains, but performs the model search only. More recently, [162] proposed a calculus for extending theory of finite sets to theory of finite relations in CVC4 solver [12], which allows relational constraint solving in SMT.

Use of abstract interpretation in satisfiability procedures

Silva et al. [86,87,89] presented an abstract interpretation account of satisfiability algorithms derived from DPLL procedure. The abstract satisfaction framework in [122] provide a lattice-theoretic characterization of satisfiability procedures and provides an unified algebraic treatment to satisfiability procedures and program analyzers. The results of [122] can be perceived from three different perspectives – 1) characterizing satisfiability procedure as abstract interpretation, 2) using satisfiability procedures to refine static analysis, and 3) lifting solver and deduction algorithms to new logics. This enables technology transfer between abstract interpretation community and satisfiability research that go beyond black-box integration approach. In this dissertation, we instantiate the lattice-theoretic characterization of CDCL for safety verification of hardware designs.

Abstract domains

The commonly used library for numerical abstract domains is the APRON C library [131]. This library is used for the static analysis of the numerical variables of a program by abstract interpretation. APRON provides a C API interface to various abstract domains and libraries such as *BOX*, *OCTAGON*, *Convex Polyhedra* and *Linear Equalities* library. Such analysis aims to compute invariants over numerical variables in the program [17]. Our domain implementation in Section 7.3.1 supports a precise complementation operation. Standard abstract interpretation does not require a complementation operator, so abstract

domain libraries, such as APRON, do not provide one. ACDLS can be instantiated over a combination of abstract domains, known as Product domain, some of which may not admit a precise complementation property. Hence, decision and learning may take place over domains that admit precise complements and deductions may be performed over domains that do not satisfy a precise complementation property. However, in this dissertation, we instantiate ACDLS over those domains only that admits precise complements.

Refinement procedures

Techniques such as Counterexample Guided Abstraction Refinement (CEGAR) [58] perform automatic refinement of the underlying abstraction through abstract counterexamples. On the other hand, refinement approaches in abstract interpretation can be classified into two types – 1) refine merging or join operations through dynamic trace partitioning [182], or 2) refine widening through strategy iterations [102] or leveraging decision procedures [166]. Other kinds of refinement in abstract interpretation include the works of [118, 119], which perform domain refinement, that is, changes the abstract domain during the analysis, similar to CEGAR. Techniques such as Symbolic Trajectory Evaluation (STE) [41, 211] is also based on abstraction and refinement approaches for verifying complex hardware circuits containing large data paths. STE is applied to circuit which is described as a graph over nodes, where each node can be a gate or a latch. Specification in STE consists of assertions in a restricted temporal language over 4-valued truth domain $\{0,1,X,\perp\}$ and is of the form $A \rightarrow C$, where antecedent A expresses the contraints on nodes n at time t and the consequent C expresses requirements that must hold on node (n,t). The initial abstraction in STE is derived from the given specification which initializes all inputs not appearing in A to the unknown value (X), while the rest of the inputs are initialized according to the constraints in A to the values 0 or 1 or to symbolic variables. By contrast, abstraction in ACDLS is computed by analyzing the execution of the whole system over a given abstract domain. The refinement in STE amounts to changing the assertion in order to compute the node values more accurately. The refinement is an iterative process, which terminates when the result of evaluating the assertion on the circuit give a truth value other than X. Note that the truth value X indicates that the antecedent A is too coarse and underspecifies the circuit.

In contrast to the previous refinement approaches, ACDLS performs transformer refinement by learning new abstract transformers from the conflict which corresponds to the partial safety proof (see Section 6.9.2). However, ACDLS does not perform domain refinement, that is, ACDLS chooses an abstract domain upfront and continues the analysis with the same domain without changing it.

Differences and improvements over abstract satisfaction

The instantiation of abstract satisfaction [87] for program analysis [88] is strictly restricted to analysis of numerical programs only. Hence, the only supported abstract domain is Interval domain. However, for analysis of systems that require relational domains or Product domains, the technique proposed in [88] is inadequate. The analysis of [88] is strongly tied to the CFG and hence the decision and learning are restricted to the initial control-flow nodes of the CFG, while supporting deduction through forward analysis only. Also, [88] does not support Unique Implication Point (UIP)-based learning [21].

In this dissertation, we address the above limitations. ACDLS is primarily used for bounded safety verification of software netlist designs generated from the hardware RTL designs. By contrast, the purpose of [88] was to detect a specific class of bugs for numerical programs, namely, overflow or undeflow errors and Interval domain is sufficient to detect these kinds of error. However, ACDLS is instantiated over a Template-based abstract domain which contains Booleans, Intervals, Octagons, Zones, Equality and fixed-coefficient Polyhedra domain. The abstract transformers in ACDLS are implemented using symbolic formulas (see Section 7.6.2), similar to Reps et al. [181], which gives precise results. ACDLS also supports forward, backward and multi-way analysis. Furthermore, the analysis using ACDLS operates on a logical encoding of the system which is represented by Static Single Assignment form. Hence, decisions and learning in ACDLS are not tied to control locations in contrast to [88]. Also, conflict analysis in ACDLS is performed via UIP computation (see Section 7.7).

6.3 Definitions

We define the notion of meet irreducibles in a lattice, γ-complete transformer, precise complementation property of lattice, downsets and upwards interpolation. These concepts are used later for explaining Abstract Conflict Driven Learning for Safety.

Definition 6.3.1. (Meet Irreducible) A *meet irreducible m* in a complete lattice structure L is an element with the following property.

$$\forall m_1, m_2 \in L:\ m_1 \sqcap m_2 = m \implies (m = m_1 \vee m = m_2),\ m \neq \bot \tag{6.1}$$

Intuitively, a meet irreducible m is an element in the lattice L such that m cannot be expressed as the meet of two other elements in L. For the Interval lattice, the meet irreducible elements are of the form $\langle x : [min, v]\rangle$ or $\langle x : [v, max]\rangle$, where v may be any integer value, and *min*, *max* are the minimum and the maximum values of integer, respectively.

Definition 6.3.2. (γ-complete). Let $(P,\subseteq)$ and $(Q,\sqsubseteq)$ be two posets which forms a Galois connection, denoted by $(P,\subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (Q,\sqsubseteq)$. Let the transformer $g\colon Q \to Q$ soundly approximate $f\colon P \to P$. The transformer g is γ-complete at element $b \in Q$ if $f(\gamma(b)) = \gamma(g(b))$. The transformer g is γ-complete over poset Q if it is γ-complete at every element of Q. Intuitively, if g is γ-complete at $b \in Q$, then it is maximally precise at b and thus every abstract transition from b also exists in P. This suggest that there are no spurious transitions from b. The notion of γ-completeness is presented by Giacobazzi et al. [105].

Definition 6.3.3. (Precise Complement) Consider an element a in a complete lattice L. An element $\bar{a} \in L$ is called *precise complement* of the element $a \in L$ iff the complementation $(\neg)$ of the concretization of $\bar{a}$ equals the concretization of a, that is, $\neg\gamma(\bar{a}) = \gamma(a)$ holds.

Definition 6.3.4. (Downsets) Let $(P,\sqsubseteq)$ be a poset. The set $Q \subseteq P$ is a *downset* or *downwards closed*, denoted by $\downarrow Q$, if $Q = \{p \in P \mid \exists q \in Q.p \sqsubseteq q\}$. We denote the set of all downsets of P by $\mathbb{D}(P)$.

Definition 6.3.5. (Downset Completion) For a concrete domain $(C,\subseteq,\cup,\cap)$, let $(A,\sqsubseteq,\sqcup,\sqcap)$ be an abstraction of the concrete domain w.r.t. a Galois connection (α,γ). A *downset completion* of a lattice A is a complete lattice $\mathbb{D}(A)$ that is equipped with disjunction and is an underapproximation of the concrete domain defined through the following abstraction and concretization function.

$$\alpha_{\mathbb{D}(A)}(P) \triangleq \{a \mid \gamma(a) \subseteq P\} \quad \gamma_{\mathbb{D}(A)}(Q) \triangleq \bigcup\{\gamma(a) \mid a \in Q\}$$

Definition 6.3.6. (Upward Interpolation) An *upward interpolation* on a lattice L is a function $int\uparrow\colon L \times L \to L$ such that $a \sqsubseteq b \implies a \sqsubseteq int\uparrow(a,b) \sqsubseteq b$ for all $a,b \in L$.

6.4 Program Model

Let *Prog* be a program. Let *Var* be the set of variables in the program *Prog* and let *Val* be the set of values that each variable can take and may be a scalar number in set of signed or unsigned integers $\mathbb{Z}$ of width N bits, where N can be 32-bit or 64-bit for instance. We consider *programs* with bounded loops and finite recursion depth along with safety properties given as a set of assertions, *Assn*, in the program. All the loop bounds in the program are known a-priori. All functions calls in the program are inlined before analysis.

6.5 Semantic Representations of Program

The *concrete semantics* of a program is the most precise mathematical description of the program behavior. Any other semantic representation of a program obtained through static analysis such as *data-flow analysis* [168] or *set-constraint* [73] based analysis is an abstract semantics, which is derived from the concrete semantics via Galois connections.

Several semantic representations [67, 75] of a program have been proposed in the literature to analyze program properties. Cousot [75] defines a hierarchy of abstraction semantics of a state transition system M corresponding to a control-flow graph G. Cousot starts from partial trace semantics and derives successive approximations via Galois connections, which follows the sequence – a) *partial trace semantics*, b) *reflexive transitive closure semantics*, c) *reachability semantics*, d) *interval semantics*. Each semantic representation differs from the other in the precision of the information. Abstract semantics are less precise than their concrete counterparts and hence can prove less program properties, but are cheaper to compute or approximate. Cousot in [67] gives a complete range of abstract semantics of a program.

Partial Trace Semantics

The semantics of a program computes the *states* of a program which gives a concise mathematical meaning of a program. The collecting semantics of a program computes all possible memory states that can occur during the execution of a program. Recall that a trace π of a state transition system $M = (\Sigma, \mathcal{R}, I)$ is a finite sequence of states that follows the transition relation $\mathcal{R}$. Let Σ_M^n denote the set of finite execution traces of length n. Then, $\Sigma_M^0 = \emptyset$, while $\Sigma_M^1 = \{s \mid s \in \Sigma\}$. Further, a trace of length $(n+1)$ can be expressed recursively as $\Sigma_M^{(n+1)} = \{\sigma s s' \mid \sigma s \in \Sigma_M^n \wedge (s, s') \in \mathcal{R}\}$.

We will denote the *partial trace semantics* as $[\![M]\!]_t$ which is the powerset of traces, $(\mathbb{P}(\Pi), \subseteq)$. A fixed point characterization of the trace semantics is defined as follows.

$$lfp\ F_M^t \text{ where } F_M^t(X) \hat{=} \{s \mid s \in \Sigma\} \cup \{\sigma s s' \mid \sigma s \in X \wedge (s, s') \in \mathcal{R}\}$$

Collecting Semantics over Traces

The collecting semantics [75] of M, Σ_M^*, is the set of all such finite execution traces of transition system M, and is given by $\Sigma_M^* \hat{=} \bigcup_{n \geq 0} \Sigma_M^n$.

Reachability Semantics

Recall from Section 2.4 that the operational semantics of a CFG $G = (V, E, \mathcal{S}, \mathcal{T}, I)$ can be represented using a state transition system M. A *reachability* semantics [163] of a state transition system $M = (\Sigma, \mathcal{R}, I)$ is the set of states that are reachable from the initial states I. A reachability semantics is a complete lattice of powerset of states, defined as $(\mathbb{P}(V \times (Var \rightarrow Value)), \subseteq, \cup, \cap)$ and is denoted by $[\![M]\!]_r$. A fixed point characterization of the reachability semantics is defined as follows.

$$lfp\ F^r_M \text{ where } F^r_M(S) \hat{=} I \cup \{s' \mid \exists s \in S \wedge (s, s') \in \mathcal{R}\}$$

Cousot in [75] shows that a reachability semantics can be derived from a partial trace semantics through a sequence of abstractions. Thus, the trace semantics $[\![M]\!]_t$ is more precise than the reachability semantics $[\![M]\!]_r$. However, it is practically infeasible to compute the trace semantics of a program that finds all memory states which occur during a program execution. We now present various transformers that operate over program traces and define the safety verification problem.

6.6 Trace Transformers and CFG Safety

The semantics of a state transition system M corresponding to a control-flow graph G can be defined in terms of *state semantics* [75] or *trace semantics* [75]. A state semantics ($[\![M]\!]_s$) of M is the same as the reachability semantics. A trace semantics ($[\![M]\!]_t$) of M is the set of well-formed traces of M. In Section 6.11, we define various classical *state transformers* of $M = (\Sigma, \mathcal{R}, I)$ that operate over the powerset of states, $\mathcal{P}(\Sigma)$. We now define various trace transformers that operate over $\mathbb{P}(\Pi)$ and formulate *CFG safety* in terms of these transformers. The term $\sigma_i \in \Sigma$ refers to a state and $\pi \in \Pi$ ranges over a set of traces in Σ^*.

Definition 6.6.1. (Strongest Postcondition) A *strongest postcondition* transformer, $tpost(T)$, of a set of traces $T \subseteq \Pi$ is defined as $tpost(T) \hat{=} \{\pi\sigma_k\sigma_l \mid \exists \pi\sigma_k \in T \wedge \sigma_k \rightarrow \sigma_l\}$. Here, the trace $\pi.\sigma_k$ is extended with its postcondition state σ_l.

Definition 6.6.2. (Weakest Precondition) A *weakest precondition* transformer, $\widehat{tpre(T)}$, of a set of traces $T \subseteq \Pi$ is defined as $\widehat{tpre(T)} \hat{=} \{\pi\sigma_a \mid \forall \sigma_b \in \Sigma.\pi\sigma_a\sigma_b \in T \vee \sigma_a \not\rightarrow \sigma_b\}$.

Definition 6.6.3. (Existential Precondition) An *existential precondition* transformer, $tpre(T)$, of a set of traces $T \subseteq \Pi$ is defined as $tpre(T) \hat{=} \{\sigma_a\sigma_b\pi \mid \exists \sigma_b\pi \in T \wedge \sigma_a \rightarrow \sigma_b\}$. Here, the trace $\sigma_b\pi$ is extended with its precondition state σ_a.

Definition 6.6.4. (Universal Postcondition) A *universal postcondition* transformer, $\widehat{tpost(T)}$, of a set of traces $T \subseteq \Pi$ is defined as $\widehat{tpost(T)} \hat{=} \{\sigma_k\pi \mid \forall \sigma_l \in \Sigma.\sigma_l\sigma_k\pi \in T \vee \sigma_l \not\rightarrow \sigma_k\}$.

CFG Safety

Let $G = (V, E, \mathcal{S}, \mathcal{T}, I)$ be a CFG with a special error node $\Xi \in V$. A trace $\pi \in G$ is safe if it does not terminate in the error location. A CFG G is *safe* with respect to Ξ if all traces π of G are safe with respect to Ξ.

We define two trace transformers over $\mathbb{P}(\Pi)$ for safety checking of a CFG, an unsafe trace transformer, f_{unsafe}, and a safe trace transformer, f_{safe}. A CFG G is *safe* exactly if $f^G_{unsafe}(\pi) = \emptyset$, $\forall \pi \in \mathbb{P}(\Pi)$. Recall that a well-formed trace begin with an initial state and follow the transition relation (see Definition 2.4.8 in Section 2.4.8).

$$f^G_{unsafe}(T) \hat{=} \{\pi \mid \pi \in T \wedge \pi \text{ is well formed and not safe}\} \tag{6.2}$$

$$f^G_{safe}(T) \hat{=} \{\pi \mid \pi \in T \vee \pi \text{ is not well formed or safe}\} \tag{6.3}$$

Fixed point Characterization of Unsafe Trace Transformer

We present a fixed point characterization of the unsafe trace transformer that finds a counterexample in a CFG. Note that a counterexample is a trace in $\mathbb{P}(\Pi)$ that terminates in an error location Ξ. Let χ denote the set of states $\{(v, \omega) \mid v = \Xi\}$ that are at the error location. Then, the unsafe trace transformer of CFG G, can be characterized as follows.

Proposition 6.6.1. $f^G_{unsafe}(T) = T \cap (lfp\ Z.\ I \cup tpost(Z)) \cap (lfp\ Z.\ \chi \cup tpre(Z))$

The fixed point characterization of unsafe trace transformer is described as follows. Let $C1 = (\mathsf{lfp} Z.\ I \cup tpost(Z))$ and $C2 = (\mathsf{lfp} Z.\ \chi \cup tpre(Z))$. $C1$ gives the set of traces that start from an initial state I and follow the transition relation $\mathcal{R}$. $C2$ gives the set of traces that follow the transition relation and terminate in an error state. The set $C1 \cap C2$ is the set of well-formed traces that terminate in an error state. It follows that $f^G_{unsafe}(T) = T \cap C1 \cap C2$.

Note that the two fixed points above, for computing $C1$ and $C2$, represents a forward and backward analysis respectively. Combination of forward and backward analysis provide strictly greater precision than applying either in isolation in the abstract domain [74]. In the context of model checking, Govindaraju et al. [113] and Cabodi et al. [43] combines the approximate forward analysis with the exact backward analysis for efficient hardware verification.

The fixed point characterization of the concrete safe trace transformer of the CFG G is given as follows.

Proposition 6.6.2. $f^G_{safe}(T) = T \cup (gfp\ Z.\ \neg\chi \cap \widehat{tpre(Z)})$

Properties of trace transformers of $\mathbb{P}(\Pi)$

We describe the properties of the various trace transformers of $\mathbb{P}(\Pi)$.

Proposition 6.6.3. A transformer f^G_{unsafe} on $(\mathbb{P}(\Pi), \subseteq, \cap, \cup)$ is completely additive and reductive exactly if there is some $x \in \mathbb{P}(\Pi)$ such that for any $y \in \mathbb{P}(\Pi), f^G_{unsafe}(y) = y \cap x$.

Proof. Assume f^G_{unsafe} is completely additive and reductive. It follows that $y \cap f^G_{unsafe}(\top) = y \cap f^G_{unsafe}(y \cup \neg y)$. The additive property of f^G_{unsafe} gives $y \cap (f^G_{unsafe}(y) \cup f^G_{unsafe}(\neg y))$. By distributivity, we get $(y \cap f^G_{unsafe}(y)) \cup (y \cap f^G_{unsafe}(\neg y))$. Due to reductive property, $f^G_{unsafe}(\neg y) \subseteq \neg y$, so we can write $(y \cap f^G_{unsafe}(\neg y)) = \bot$ and $(y \cap f^G_{unsafe}(y)) = f^G_{unsafe}(y)$. Hence, $y \cap f^G_{unsafe}(\top) = f^G_{unsafe}(y)$, which follows that the element $f^G_{unsafe}(\top)$ is x.

Let us consider a function $f^G_{unsafe}(y) = y \cap x$ for some $x \in \mathbb{P}(\Pi)$. Clearly, f^G_{unsafe} is reductive. Let $\Pi^\# \subseteq \Pi$, then $f^G_{unsafe}(\bigcup \Pi^\#) = (\bigcup \Pi^\# \cap x)$. By distributivity, we can write $\bigcup_{\pi \in \Pi^\#}(\pi \cap x) = \bigcup_{\pi \in \Pi^\#} f(\pi)$. □

We now show that the transformer f^G_{unsafe} is closure operator.

Proposition 6.6.4. Every completely additive and reductive transformer $f^G_{unsafe} : \mathbb{P}(\Pi) \to \mathbb{P}(\Pi)$ is lower closure operator.

Proof. We show that f^G_{unsafe} is monotonic and idempotent. To show f^G_{unsafe} is monotonic or order-preserving, let us consider $\pi, \pi' \in \Pi$ such that $\pi \subseteq \pi'$. By proposition 6.6.3, it holds that $f^G_{unsafe}(\pi) = \pi \cap f^G_{unsafe}(\top)$ and $f^G_{unsafe}(\pi') = \pi' \cap f^G_{unsafe}(\top)$, hence $f^G_{unsafe}(\pi) \subseteq f^G_{unsafe}(\pi')$. To show f^G_{unsafe} is idempotent, we have $f^G_{unsafe}(f^G_{unsafe}(\pi)) = f^G_{unsafe}(\pi \cap f^G_{unsafe}(\top)) = \pi \cap f^G_{unsafe}(\top) \cap f^G_{unsafe}(\top) = \pi \cap f^G_{unsafe}(\top) = f^G_{unsafe}(\pi)$. □

The De Morgan dual of a completely additive and reductive transformer is multiplicative and extensive. Therefore, dually every multiplicative, extensive transformer over $\mathbb{P}(\Pi)$ is upper closure.

Proposition 6.6.5. Let $f^G_{unsafe} : \mathbb{P}(\Pi) \to \mathbb{P}(\Pi)$ be a completely additive and reductive transformer over $\mathbb{P}(\Pi)$. Then $f^G_{safe} = \neg \circ f^G_{unsafe} \circ \neg$ is a completely multiplicative and extensive transformer over $\mathbb{P}(\Pi)$.

Proof. If $f^G_{unsafe} : \mathbb{P}(\Pi) \to \mathbb{P}(\Pi)$ is completely additive and reductive transformer over $\mathbb{P}(\Pi)$, then we have $f^G_{unsafe}(\pi) = \pi \cap f^G_{unsafe}(\top)$ for any $\pi \in \mathbb{P}(\Pi)$. It also holds for any $\pi \in \mathbb{P}(\Pi)$ that, $f^G_{safe}(\pi) = \neg f^G_{unsafe}(\neg\pi) = \neg(\neg\pi \cap f^G_{unsafe}(\top)) = \pi \cup \neg f^G_{unsafe}(\top)$. From the duality of proposition 6.6.3, we have that f^G_{safe} is completely multiplicative and extensive. □

Proposition 6.6.6. An additive and reductive transformer $f^G_{unsafe}(\pi) = \bot \; \forall \pi \in \mathbb{P}(\Pi)$ exactly if its De Morgan dual, which is multiplicative and extensive transformer $f^G_{safe}(\pi) = \top \; \forall \pi \in \mathbb{P}(\Pi)$.

Informally, the transformer $f^G_{unsafe}(\Pi^\#)$ for $\Pi^\# \in \mathbb{P}(\Pi)$ can be seen to be removing elements from the input set $\Pi^\#$, whereas $f^G_{safe}(\Pi^\#)$ can be seen to be adding elements to the input set. For example, f^G_{unsafe} can start with the set of all traces and iteratively removes elements that are not safe from the input set. On the other hand, f^G_{safe} starts from an empty set and iteratively add all safe traces to the set. Imagine a program with two traces, trace corresponding to if branch is π_1 which is safe and trace corresponding to *else* branch is π_2 which is unsafe. Now, $f^G_{safe}(\pi_1 \cup \pi_2) = f^G_{safe}(\pi_1) \cup f^G_{safe}(\pi_2) = \bot \cup \pi_2 = \pi_2$. Hence, $f^G_{safe}(\pi_1 \cup \pi_2) \subseteq (\pi_1 \cup \pi_2)$. Clearly, f^G_{safe} is reductive in this case.

6.7 Approximations of Trace Semantics

Figure 6.1 shows the sequence of syntactic translation steps and semantic approximations of concrete set of traces, $\mathbb{P}(\Pi)$. Recall that a program is represented by a Control-flow Graph $G = (V, E, S, \mathcal{T}, I)$ (see Section 2.4), marked by the syntactic translation step $\mathcal{T}_1$ in Figure 6.1. Classical abstract interpretation interprets a CFG as equation system [88, 163, 189], which is described in Section 6.7.1. The box in *red* gives the corresponding lattices that an abstract interpreter operates on. The ACDLS procedure presented in this dissertation operates on a different lattice which is obtained by syntactic translation step $\mathcal{T}_2$ that translates a CFG to Static Single Assignment (SSA) form [77, 78]. The bounded box in *green* shows the corresponding lattices that ACDLS operates on.

We first describe an equation system that is derived from a CFG which is commonly used for abstract interpretation. Later, we describe a syntactic translation of CFG to Static Single Assignment form which is used by ACDLS procedure. In Section 6.8, we describe the corresponding lattices over these structures and various transformers that operate on these lattices. The translation steps $\mathcal{T}_1$ and $\mathcal{T}_1'$ in Figure 6.1 are trivial and they are very well-known in the context of compilers, so we do not explain these steps.

6.7.1 CFG to Equation Systems

Recall that an operational semantics of a CFG G can be defined using a state transition system. An equation system [163] can be derived from a state transition system, which introduces a set-valued variable $(S_v)_{v \in V}$ for each program location that takes values in the powerset of environments, $\mathbb{P}(Var \mapsto Value)$. The variables S_v are related through

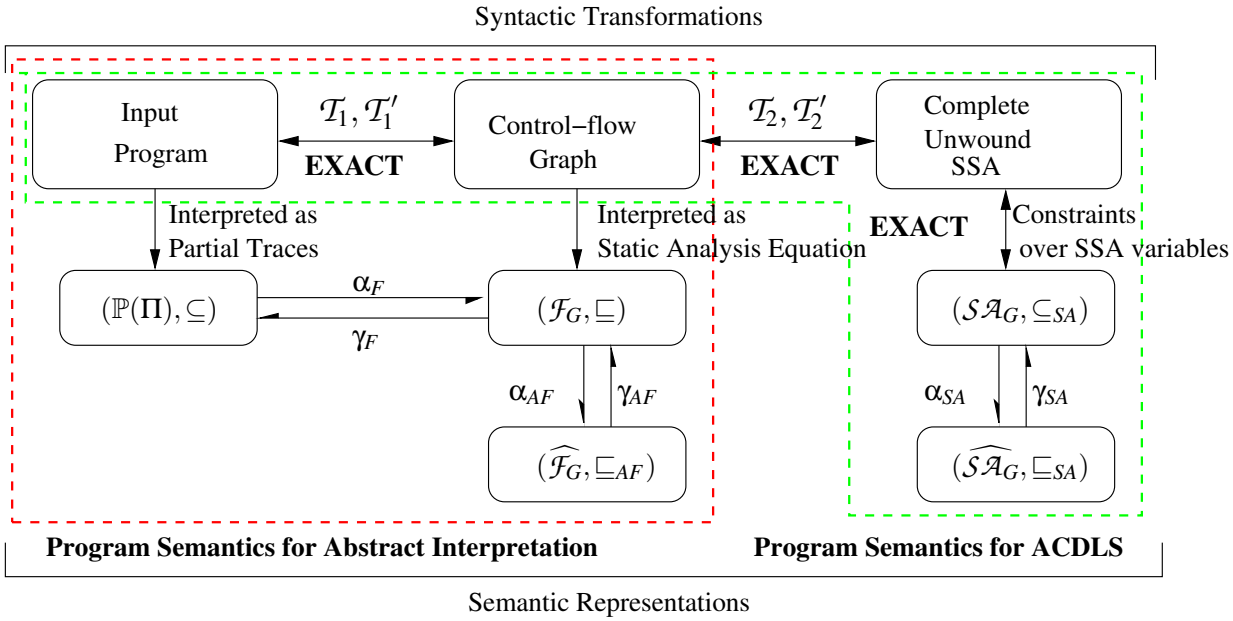


Figure 6.1: Semantic representation of program

the transformers associated with program statements that express the data-flow equations between individual control-flow nodes in G. We will call this equation system, *static analysis equation*. The solution to these equations yields the data-flow information of G. The resulting lattice that describes these static analysis equations is given by $(\mathcal{F}_G, \sqsubseteq)$. Similar equation systems are described by Vergauwen and Lewi [215] and Mader [154, 155], known as Boolean Equation Systems (BESs), for model checking modal μ-calculus formulae. Later, Groote et al. [117] extended BES with data, known as Parameterised Boolean Equation Systems (PBESs), which are sequences of the form $\sigma X(d_1 : D_1, \ldots, d_n : D_n) = \varphi$, where σ is either a least or greatest fixed point symbol, d_i is a data variable of type D_i, X is a predicate variable that depends on data variables d_i and φ is a predicate formula. The PBESs are used for model checking processes with (arbitrary) data.

Figure 6.2 gives the CFG representation of a program (on the left) and its corresponding static analysis equations (on the right). The function $post(S)$ computes the successor state of a set of states S that can be reached in one step. A set-valued variable S_v is introduced at every control location v in the CFG. Note that the variables S_v are related through the postcondition transformer *post* associated with the program statements, for example, the variable S_{n6} at the loop head merges the control-flow from S_{n5} and S_{n7}. The CFG is *safe* if S_{Error} is empty $(\emptyset)$.

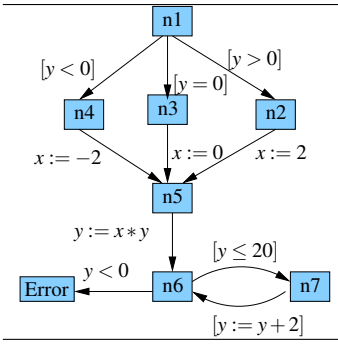
Control-Flow Graph	Static Analysis Equation
	$S_{n1} = \top,$ $S_{n2} = post_{y>0}(S_{n1})$ $S_{n3} = post_{y=0}(S_{n1})$ $S_{n4} = post_{y<0}(S_{n1})$ $S_{n5} = post_{x:=2}(S_{n2}) \cup post_{x:=0}(S_{n3}) \cup post_{x:=-2}(S_{n4}),$ $S_{n6} = post_{y:=x*y}(S_{n5}) \cup post_{y:=y+2}(S_{n7}),$ $S_{n7} = post_{y\leq 20}(S_{n6}),$ $S_{Error} = post_{y<0}(S_{n6})$

Figure 6.2: A CFG and its static analysis equation

Collecting Semantics over Equation Systems

A collecting semantics over static analysis equations of a program gathers for each program variable and program location its value during program execution.

6.7.2 Logical Encoding of Program Semantics

Program Transformation $\mathcal{T}_2$

The program transformation $\mathcal{T}_2$ shown in Figure 6.1 involves two separate steps.

1. Generation of a *bounded program*
2. Translation of the acyclic control-flow of a bounded program to SSA

Generation of Bounded Program

Recall from Section 6.4 that we consider programs with bounded loops and finite recursion depth in this dissertation. A *bounded program* is obtained from an input program by a transformation that unfolds loops and recursions completely, which generates an acyclic control-flow structure. A complete unrollings of all loops in a program means that every loop is unwound up to its maximum bound. However, not every program can be completely unwound, for example, programs with unbounded loops. For such programs, we unfold loops and recursions a finite number of times in this dissertation. Figure 6.3 and Figure 6.4 gives an input program and the corresponding bounded program obtained through loop unrolling, respectively. The CFG representation of the bounded program is translated into a SSA form. The translation from CFG to SSA is a well-known technique in optimizing compiler and verification [55, 77, 78, 184]. We only briefly explain the details of the CFG to SSA translation since they are standard.

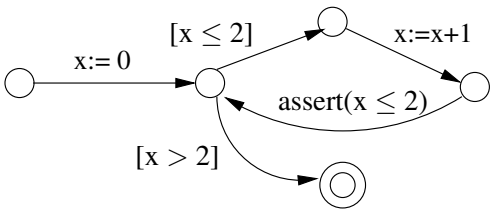


Figure 6.3: Input program with bounded loops

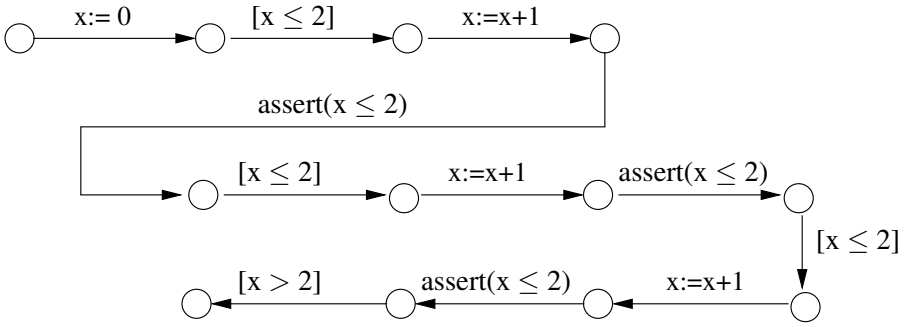


Figure 6.4: Generating a Bounded Program throug loop unrolling

CFG to Static Single Assignment Form

The translation from CFG to SSA follows two separate steps. The first step gives a unique *index* to each definition of a program variable, and each use of that variable is given the index of the definition that reaches it; the second step inserts special ϕ-functions at control-flow join points where a given variable may have more than one reaching definition. The argument to a ϕ-function is the set of all indexed instances of the variable that could reach the join point. Based on the current execution trace, the ϕ-function select an appropriate instance of the variable and assign it to a new instance of the variable. The translation involves the following steps.

1. A unique name for each definition point in the procedure, shown in Figure 6.5.
2. Identifies points in the procedure that merge different values from distinct control-flow paths, shown in Figure 6.5
3. Identification of induction variables in loops becomes easy by inserting a ϕ-function for any variable that is modified inside the loop [104].

Figure 6.6 gives an example of SSA translation for the bounded program in Figure 6.4.

Collecting semantics over SSA

A collecting semantics over SSA of a bounded (loop-free) program gathers for each SSA variable its value during program execution.

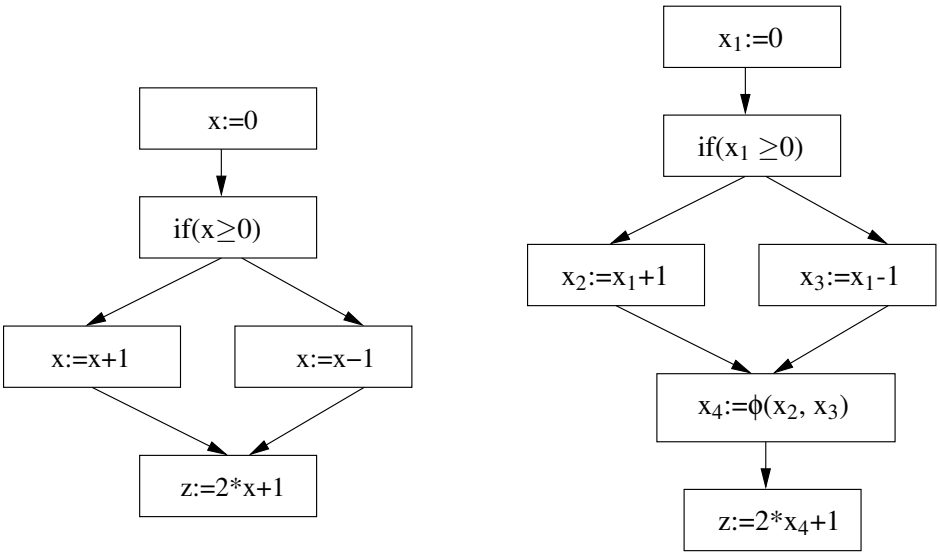


Figure 6.5: An example for CFG to SSA translation step

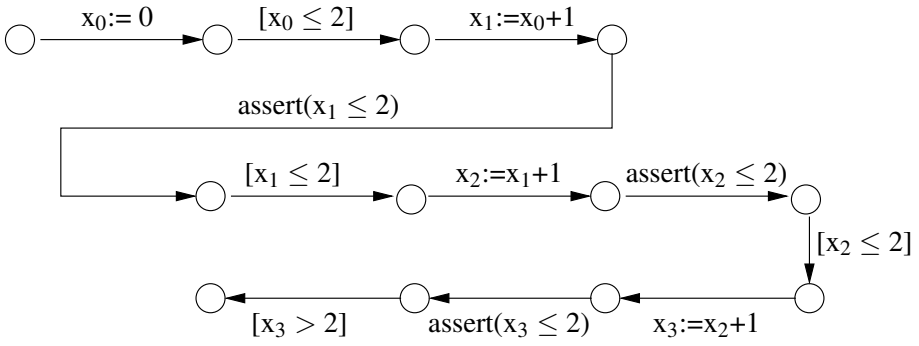


Figure 6.6: Static Single Assignment form for bounded program of Figure 6.4

Exact SSA semantics

An SSA semantics is *exact* if it is constructed from a bounded program. Recall from Section 6.7.2 that a bounded program is obtained through *complete* unwindings of all loops and recursions, which gives an acyclic code. Section 7.4.3 gives examples of *exact*, *overapproximate* and *underapproximate* SSA semantics for an input program. In this dissertation, we restrict our formalizations to complete unrollings of the bounded programs which gives an exact SSA semantics of a program.

SSA Safety

The SSA is represented by a set of constraints $\Sigma^{\dagger} = Prog \cup \{\neg \bigwedge_{a \in Assn} a\}$, where *Prog* contains an encoding of the statements in the program as constraints, obtained after translating the program into single static assignment (SSA) form [5, 78, 184]. Based on the above

program representation, we define a *safety formula* φ as the conjunction of all elements in $\Sigma^{\dagger}$, that is, $\varphi := \bigwedge_{\sigma \in \Sigma^{\dagger}} \sigma$. The formula φ is unsatisfiable if and only if the program is safe.

Example 6.7.1. The safety formula φ for the bounded program of Figure 6.6 is shown below. Note that φ is obtained by taking conjunction of all constraints and the negation of set of assertions.

$$\begin{aligned}\varphi :=&(x_0 = 0) \wedge (x_0 \leq 2) \wedge (x_1 = x_0 + 1) \wedge (x_1 \leq 2) \wedge (x_2 = x_1 + 1) \wedge \\ &(x_2 \leq 2) \wedge (x_3 = x_2 + 1) \wedge ((x_1 > 2) \vee (x_2 > 2) \vee (x_3 > 2))\end{aligned} \tag{6.4}$$

Translation from static single assignment to program trace

We will now describe the translation from SSA to program trace via the program transformation step $\mathcal{T}_2'$.

Figure 6.7: Translation from SSA to program trace

Program Transformation $\mathcal{T}_2'$

The program transformation $\mathcal{T}_2'$ shown in Figure 6.12 involves two separate steps.

1. Copy-insertion [203] to remove ϕ-nodes.
2. Variable renaming in each basic blocks.

The translation of SSA to an original program following these syntactic translation steps is a well-known technique in compiler-based code optimization [39, 78, 203]. Briggs et al. [39] give an algorithm for translation of SSA to the original program by replacing ϕ-functions with appropriately-placed *copy* instructions. The program semantics is preserved by inserting a copy statement for each argument of ϕ-function, at the end of each predecessor block of the ϕ-node. Figure 6.8 shows an example of copy insertion to replace ϕ-function in SSA. We skip the details of the translation from SSA back to original program since they are standard.

6.8 Lattice for Approximation of Trace Semantics

In this section, we describe the various lattices over the static analysis equations and static single assignment form for a given control-flow graph. Figure 6.9 shows the various lattices that approximate the concrete trace semantics.

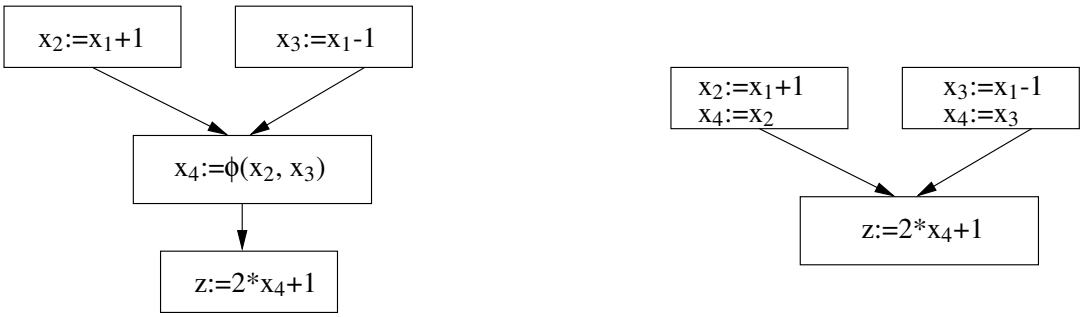


Figure 6.8: Back-translation from SSA to CFG using copy insertion

In order to lift CDCL to safety checking of programs, it is imperative to find a suitable trace-based abstraction that admits the precise complementation property. The work of [89] shows that each meet irreducible of the partial assignments domain in the CDCL-based SAT solver admits precise complement [89], which are necessary for learning from conflicts. To find a trace-based abstraction with the above property, we first show that the conventional means to represent a program through abstraction of a static analysis equation does not admit precise complements with respect to unsafe traces. In this dissertation, we represent program traces using the logical encoding of a CFG that bounded model checkers use. We will show in Section 6.8.4 that this representation allows us to generalize CDCL to safety verification over abstract lattices.

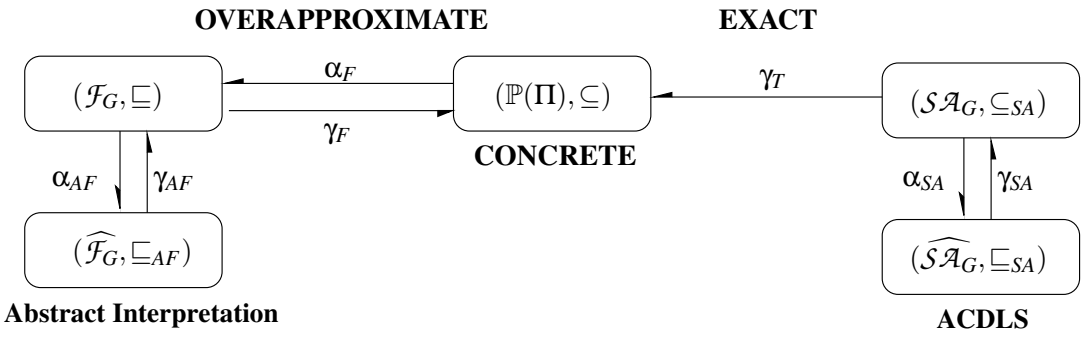


Figure 6.9: Approximation of trace semantics used by ACDLS (in the right) and abstract interpretation (in the left)

6.8.1 Concrete Flow Lattice

The abstract interpretation technique for program analysis computes a fixed-point over static analysis equations [70, 164]. The advantage of using approximations of static analysis equations over trace-based abstraction is that the former only requires abstractions of memory states.

We now define the *concrete flow lattice* that describes these static analysis equations with respect to the set of control locations.

Definition 6.8.1. (Concrete Flow Lattice). A *concrete flow lattice* corresponding to a CFG $G = (V, E, \mathcal{S}, \mathcal{T}, I)$ is a complete lattice $(\mathcal{F}_G, \sqsubseteq, \sqcap, \sqcup)$, defined as follows.

$$\mathcal{F}_G \triangleq V \to \mathbb{P}(\Omega) \qquad \forall a, b \in \mathcal{F}_G.\ a \sqsubseteq b \text{ if } \forall v \in V.\ a(v) \subseteq b(v)$$

$$a \sqcap b \triangleq v \mapsto a(v) \cap b(v) \qquad a \sqcup b \triangleq v \mapsto a(v) \cup b(v)$$

Recall from Section 2.4 that Ω is a memory state which is a mapping of program variables to its corresponding value. The concrete flow lattice is an overapproximation of the concrete powerset of traces denoted by $\mathbb{P}(\Pi)$, and there exists a Galois connection between the two as shown below.

$$(\mathbb{P}(\Pi), \subseteq, \cup, \cap) \underset{\alpha_F}{\overset{\gamma_F}{\leftrightarrows}} (\mathcal{F}_G, \sqsubseteq, \sqcup, \sqcap)$$

We define the abstraction and concretization functions between the concrete trace domain $\mathbb{P}(\Pi)$ and concrete flow lattice $\mathcal{F}_G$.

$$\alpha_F(T) \triangleq \lambda v.\{\omega \mid \exists \pi \in T.\ (v, \omega) \in \pi\}$$

$$\gamma_F(a) \triangleq \{\pi \mid \forall (v, \omega) \in \pi.\ \omega \in a(v) \wedge \pi \in \Pi_{wf}\}$$

Here, Π_{wf} is a set of well-formed trace (see Definition 2.4.8). It is easy to see that $\alpha_F \circ \gamma_F$ is reductive while $\gamma_F \circ \alpha_F$ is extensive, and the pair (α_F, γ_F) forms a Galois connection. We prove this in Proposition 6.8.1.

For the purpose of static analysis, the concrete semantics of a program's control-flow graph is overapproximated using the abstract interpretation methodology [70]. The concrete domain of memory states $\mathbb{P}(\Omega)$ is abstracted to construct an abstract program model which is then used for abstract program analysis. Most static program analyzers build an abstraction A of the memory states Ω, shown below.

$$(\mathbb{P}(\Omega), \subseteq) \underset{\alpha_{AF}}{\overset{\gamma_{AF}}{\leftrightarrows}} (\mathcal{A}, \sqsubseteq_{AF})$$

Definition 6.8.2. (Abstract Flow Lattice). An *Abstract Flow Lattice* corresponding to a CFG $G = (V, E, \mathcal{S}, \mathcal{T}, I)$ and an abstraction $\mathcal{A}$ over concrete memory states Ω is a complete lattice $(\widehat{\mathcal{F}_G}, \sqsubseteq_{AF}, \sqcap_{AF}, \sqcup_{AF})$, which is defined as follows.

$$\widehat{\mathcal{F}_G} \triangleq V \to \mathcal{A} \qquad \forall a, b \in \widehat{\mathcal{F}_G}.\ a \sqsubseteq b \text{ if } \forall v \in V.\ a(v) \sqsubseteq_{AF} b(v)$$

$$a \sqcap_{AF} b \triangleq v \mapsto a(v) \sqcap_{AF} b(v) \qquad a \sqcup_{AF} b \triangleq v \mapsto a(v) \sqcup_{AF} b(v)$$

6.8.2 Complementation in Abstract Flow Lattice

Recall the notion of meet irreducible from Definition 6.3.1. A meet irreducible of the abstract flow lattice $\widehat{\mathcal{F}_G}$ is defined as follows.

$$\sqcap_{irrd}(\widehat{\mathcal{F}_G}) \hat{=} \begin{cases} n \mapsto a & \text{if } n' = n, n' \in V \wedge a \in \sqcap_{irrd}(\mathcal{A}) \\ n \mapsto \top & \text{otherwise} \end{cases}$$

Here, a node $n \in V$ is mapped to an abstract memory state $a \in \mathcal{A}$. Let $n \mapsto \bar{a}$ be a precise complement of $n \mapsto a$, where $\bar{a}$ is a precise complement of a (assuming $\mathcal{A}$ admits precise complements). However, the two sets are not equal, that is, $\gamma_{AF}(n \mapsto a) \neq \neg\gamma_{AF}(n \mapsto \bar{a})$. The element $\neg\gamma_{AF}(n \mapsto \bar{a})$ gives the set of traces where some occurrences of n are contained in the concretization of $\gamma_{AF}(a)$. On the other hand, the element $\gamma_{AF}(n \mapsto a)$ gives the set of traces where all occurrences of n are contained in the concretization of $\gamma_{AF}(a)$. This is explained below.

$$\gamma_{AF}(n \mapsto a) \hat{=} \{\pi \in \Pi \mid \forall (n, \omega) \in \pi.\omega \in \gamma_{AF}(a)\}$$

$$\neg\gamma_{AF}(n \mapsto \bar{a}) \hat{=} \neg\{\pi \in \Pi \mid \forall (n, \omega) \in \pi.\omega \notin \gamma_{AF}(a)\}$$

$$\neg\gamma_{AF}(n \mapsto \bar{a}) \hat{=} \{\pi \in \Pi \mid \exists (n, \omega) \in \pi.\omega \in \gamma_{AF}(a)\}$$

We give an example to explain the complementation of the abstract flow lattice.

Example 6.8.1. Consider the CFG of Figure 6.2 which is analyzed over the Interval domain. Let us consider the node n_1 of the CFG and the interval for the variable y at n_1, which is denoted by $n_1 \mapsto y : [0, \infty]$. Then, $\gamma_{AF}(n_1 \mapsto y : [0, \infty])$ gives the set of traces denoted by $\Pi^{\#} = \{\{n_1, n_2, n_5, n_6, n_7, \ldots\}, \{n_1, n_3, n_5, n_6, n_7, \ldots\}\}$. The function $\gamma_{AF}(n_1 \mapsto y : [-\infty, -1])$ gives the set of traces denoted by $\{n_1, n_4, n_5, n_6, n_7, \ldots\}$, where $y : [-\infty, -1]$ is the precise complement of $y : [0, \infty]$. Now, $\neg\gamma_{AF}(n_1 \mapsto y : [-\infty, -1])$ gives the set of traces that is a subset of $\Pi^{\#}$. For example, $\neg\gamma_{AF}(n_1 \mapsto y : [-\infty, -1])$ may only contain the trace $\{n_1, n_3, n_5, n_6, n_7, \ldots\}$. Hence, some occurences of n_1 are contained in $\gamma_{AF}(y : [0, \infty])$. Thus, $\gamma_{AF}(n_1 \mapsto y : [0, \infty]) \neq \neg\gamma_{AF}(n_1 \mapsto y : [-\infty, -1])$.

The abstract control flow lattice $\widehat{\mathcal{F}_G}$ does not always have complementable meet irreducibles. So, we define a domain over logical encodings of the CFG, the elements of which have the precise complementation property.

6.8.3 Lattice Structure over SSA

We now define a lattice over SSA that admit precise complementation property. Let Var_{ssa} denote the set of SSA variables in the SSA form and $Value$ denote the set of values that these SSA variables can take. Note that the set Var_{ssa} is obtained from the CFG $G = (V, E, \mathcal{S}, \mathcal{T}, I)$ through program transformation $\mathcal{T}_2$.

Definition 6.8.3. (Static Assignment Lattice). A *Static Assignment Lattice* corresponding to a CFG $G = (V, E, \mathcal{S}, \mathcal{T}, I)$ is a complete lattice $(\mathcal{SA}_G, \subseteq_{SA}, \cup_{SA}, \cap_{SA})$, defined as follows.

$$\mathcal{SA}_G \triangleq \mathbb{P}(Var_{ssa} \to Value) \quad \forall A, B \in \mathcal{SA}_G.\ A \subseteq_{SA} B$$

$$A \cap_{SA} B \triangleq A \cap B \qquad A \cup_{SA} B \triangleq A \cup B$$

Figure 6.10 gives an example of concrete static assignment lattice over SSA variables p of type boolean and x, y of numerical types. Note that the lattice $\mathcal{SA}_G$ is a set of concrete environments over SSA variables. Intuitively, the lattice $\mathcal{SA}_G$ is a lattice of semantics, that is, the possible interpretations of a safety formula φ derived from a set of SSA constraints. The lattice $\mathcal{SA}_G$ is *not* a lattice of representations, that is, it is not the lattice of all possible SSAs. Following the result of Briggs et al. [39], the elements of $\mathcal{SA}_G$ can be mapped back to a trace of the original program. The elements marked in bold in Figure 6.10 corresponds to concrete assignments to SSA variables that maps to a concrete program trace.

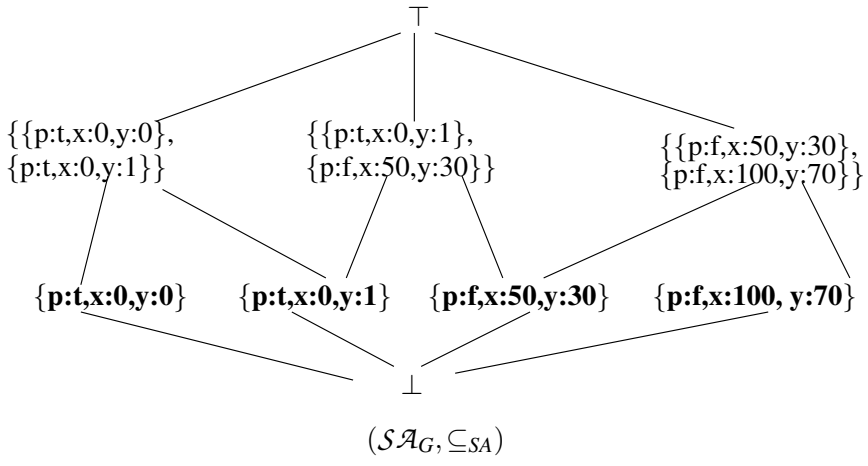


Figure 6.10: Example of concrete Static Assignment lattice over a boolean SSA variable p and two numerical SSA variables x and y that takes values in Integer domain

For convenience, we introduce a second notation to represent elements of $\mathcal{SA}_G$ lattice. For example, when we say, an element $\{x \geq 5\} \in \mathcal{SA}_G$, we mean the set of all values of x that satisfy the constraint $x \geq 5$.

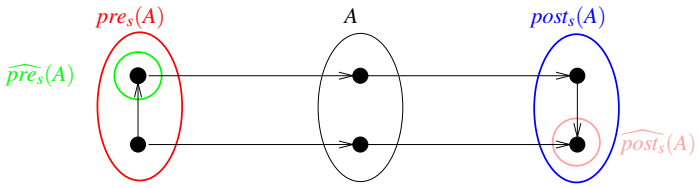


Figure 6.11: Strongest postcondition $post_s$ (Blue), Existential precondition pre_s (Red), Universal postcondition $\widehat{post_s}$ (Pink), Weakest precondition $\widehat{pre_s}$ (Green)

We now define various transformers over the $\mathcal{SA}_G$ lattice. A constraint (or state transformer) $s \in \Sigma^\dagger$ is associated with a transition relation, $\mathcal{ST}_s^\Omega$ from which we can derive classical transformers such as the strongest postcondition, existential precondition, weakest precondition and universal postcondition transformers. These transformers are defined below for a given memory state A.

Definition 6.8.4. (Strongest Postcondition and Existential Precondition).

$$post_s(A) \triangleq \{\omega' \mid \exists \omega \in \Omega.(\omega \in A) \wedge (\omega, \omega') \in \mathcal{ST}_s^\Omega\}$$

$$pre_s(A) \triangleq \{\omega \mid \exists \omega' \in \Omega.(\omega' \in A) \wedge (\omega, \omega') \in \mathcal{ST}_s^\Omega\}$$

Definition 6.8.5. (Weakest Precondition and Universal Postcondition).

$$\widehat{pre_s}(A) \triangleq \{\omega \mid \forall \omega' \in \Omega.\ (\omega' \in A) \vee (\omega, \omega') \notin \mathcal{ST}_s^\Omega\}.$$

$$\widehat{post_s}(A) \triangleq \{\omega' \mid \forall \omega \in \Omega.\ (\omega \in A) \vee (\omega, \omega') \notin \mathcal{ST}_s^\Omega\}.$$

Figure 6.11 shows the result of the application of various state transformers graphically. For the purpose of illustration, we also give the semantics of the state transformers in temporal logic with path quantifiers $(\mathcal{A}, \mathcal{E})$ that define "for all possible computations" and "for some computations" respectively, next time operator (X), and past time operator (P) [137]. A strongest postcondition $post_s(A)$ maps a set of states A to the set of all successor states that can be reached in one step. This can be expressed in temporal logic by the formula $\mathcal{E}P\,A$. An existential precondition $pre_s(A)$ maps a set of states A to the set of all states the program *may* have before executing s. This can be expressed in temporal logic by the formula $\mathcal{E}X\,A$. A weakest precondition $\widehat{pre_s}(A)$ maps a set of states A to the set of all states which can *only* reach elements of A. This can be expressed in temporal logic by the formula $\mathcal{A}X\,A$. A universal postcondition $\widehat{post_s}$ maps A to set of all states the program *must* reach after executing s. This can be expressed in temporal logic by the formula $\mathcal{A}P\,A$.

Recall from Section 6.6 that the trace transformers are defined over $\mathbb{P}(\Pi)$. The state transformers in Definition 6.8.4 and Definition 6.8.5 operate over the Static Assignment

lattice $\mathcal{SA}_G$. Furthermore, $\mathcal{SA}_G$ is an exact abstraction of $\mathbb{P}(\Pi)$ that is obtained through the syntactic translation step, $\mathcal{T}_2$. Hence, the transformers, $post_s$, pre_s, $\widehat{post_s}$, $\widehat{pre_s}$, soundly approximate $tpost$, $tpre$, $\widehat{tpost}$, $\widehat{tpre}$, respectively. The global static assignment transformers for the lattice $\mathcal{SA}_G$ over a set of constraints $\Sigma^\dagger$ can be easily derived from state transformers. This is defined next.

Definition 6.8.6. (Global Static Assignment Transformers).

$$post_{\Sigma^\dagger}, pre_{\Sigma^\dagger}, \widehat{post_{\Sigma^\dagger}}, \widehat{pre_{\Sigma^\dagger}}, : \mathcal{SA}_G \rightarrow \mathcal{SA}_G$$

$$post_{\Sigma^\dagger}(a) \hat{=} \bigcap_{\sigma \in \Sigma^\dagger} post_\sigma(a) \qquad pre_{\Sigma^\dagger}(a) \hat{=} \bigcap_{\sigma \in \Sigma^\dagger} pre_\sigma(a)$$

$$\widehat{post_{\Sigma^\dagger}}(a) \hat{=} \bigcap_{\sigma \in \Sigma^\dagger} \widehat{post_\sigma}(a) \qquad \widehat{pre_{\Sigma^\dagger}}(a) \hat{=} \bigcap_{\sigma \in \Sigma^\dagger} \widehat{pre_\sigma}(a)$$

Assuming that $tpost_G$, $tpre_G$, $\widehat{tpost_G}$ and $\widehat{tpre_G}$ are global trace transformers over CFG G, the transformers, $post_{\Sigma^\dagger}$, $pre_{\Sigma^\dagger}$, $\widehat{post_{\Sigma^\dagger}}$, $\widehat{pre_{\Sigma^\dagger}}$ soundly approximate their concrete counterparts respectively. This is because $\mathcal{SA}_G$ is an exact abstraction of $\mathbb{P}(\Pi)$. Hence, transformers that operate on $\mathbb{P}(\Pi)$ soundly approximate the transformers that operate on $\mathcal{SA}_G$.

We now define an abstraction of static assignment lattice which we call *abstract static assignment* domain. Recall from Definition 6.8.3 that the concrete static assignment domain is a complete lattice of concrete environments over SSA variables. The abstractions of concrete static assignment domains fall into two categories; an abstraction that preserves the relationship between SSA variables and the other which does not preserve any relation.

First, we give an abstraction of the static assignment lattice that does not preserve relationship between SSA variables. Then, we present an abstraction using a *template-based abstract domain* that can express relational as well as non-relational facts involving the SSA variables.

Concretizing Static Single Assignment Semantics to Program Traces Semantics

Cousot and Cousot in [72] propose a relaxation of Galois connection framework that allows to work only with a concretization operator γ or dually an abstraction operator α. We will use this concept to establish the relationship between $(\mathbb{P}(\Pi), \subseteq)$ and $(\mathcal{SA}_G, \subseteq_{SA})$ only through a concretization operator, γ_T. Figure 6.12 illustrates that γ_T can be achieved through the syntactic translation steps, $\mathcal{T}_1'$ and $\mathcal{T}_2'$, which are described in Section 6.7.2.

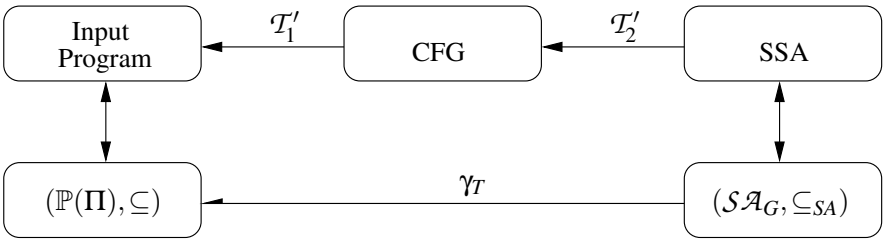


Figure 6.12: Concretization-based approximation of trace semantics in ACDLS

Definition 6.8.7. (Concretization)

$$\gamma_T : (\mathcal{SA}_G,\subseteq_{SA}) \xrightarrow{\gamma_T} (\mathbb{P}(\Pi),\subseteq)$$

Recall that $(\mathcal{SA}_G,\subseteq_{SA})$ is obtained from $(\mathbb{P}(\Pi),\subseteq)$ through syntactic translation steps, $\mathcal{T}_1$ and $\mathcal{T}_2$, hence $(\mathcal{SA}_G,\subseteq_{SA})$ is an exact abstraction of $(\mathbb{P}(\Pi),\subseteq)$ for the programs considered in this dissertation. For $A \in \mathcal{SA}_G$ which is an exact abstraction of $\pi \in \Pi$, the concretization operator $\gamma_T(A) = \pi$ is automatically monotonic.

Abstract Static Assignment Lattice over Non-relational Domains

An abstraction of $\mathcal{SA}_G$ over a non-relational domain requires each SSA variables in SSA_{var} to be abstracted independently of each other. The concrete domain of $\mathbb{P}(Var_{ssa} \to Value)$ is abstracted through a function which maps each SSA variable to an abstract value. We assume that there is a Galois connection (α_v,γ_v) such that $(Value,\subseteq) \underset{\alpha_v}{\overset{\gamma_v}{\leftrightarrows}} (\widehat{Value},\sqsubseteq)$. Then, an abstraction of $\mathcal{SA}_G$ over non-relational domain, denoted by $\mathcal{SA}_G^\dagger = Var_{ssa} \to \mathbb{P}(\widehat{Value})$, can be constructed using the following Galois connection $(\alpha^\dagger,\gamma^\dagger)$.

$$(\mathcal{SA}_G,\subseteq_{SA}) \underset{\alpha^\dagger}{\overset{\gamma^\dagger}{\leftrightarrows}} ({\mathcal{SA}_G}^\dagger,\sqsubseteq^\dagger)$$

$$\alpha^\dagger(c) \hat{=} \lambda x.\alpha_v(\bigcup\{\tau(x) \mid \tau \in c\})$$

$$\gamma^\dagger(a) \hat{=} \{\lambda x.v' \mid v' \in (\gamma_v \circ a)(x)\}$$

Example 6.8.2. An example of $({\mathcal{SA}_G}^\dagger,\sqsubseteq^\dagger)$ over an Interval domain Itv maps each SSA variable to an Interval, where $(\alpha_v,\gamma_v) = (\alpha_{Itv},\gamma_{Itv})$. Consider an SSA $c = \{x_1 := 2, x_2 \geq 5\}$, then $\alpha^\dagger(c) = \{x_1 = [2,2], x_2 = [5,\infty]\}$

Abstract Static Assignment Lattice Over Template-based Domains

Folk wisdom in the abstract interpretation community is that an analysis using non-relational domains are effective but imprecise. However, for practical purposes, it is imperative

to keep track of relations between program variables. A relational domain can express relationship between variables, though with varying expressivity, depending on a weak or strong relational domain. We now construct an abstraction of $\mathcal{SA}_G$ using a template-based domain, denoted by $\widehat{\mathcal{SA}_G}$, which can be instantiated with arbitrary templates over relational or non-relational domains. Note that from this point onwards, we will operate on $\widehat{\mathcal{SA}_G}$ instead of ${\mathcal{SA}_G}^{\dagger}$.

Definition 6.8.8. (Abstract Static Assignment Lattice). An *Abstract Static Assignment Lattice* corresponding to a CFG $G = (V, E, \mathcal{S}, \mathcal{T}, I)$ and a set X of numerical values of vector $\vec{x}$ of variables in the corresponding SSA, is a complete lattice $(\widehat{\mathcal{SA}_G}, \sqsubseteq_{SA}, \sqcap_{SA}, \sqcup_{SA})$, which is defined as follows.

$$\widehat{\mathcal{SA}_G} \triangleq C\vec{x} \leq \vec{d}, \text{ where } \vec{d} \text{ is a constant vector and C is a coefficient matrix}$$

The abstraction and concretization functions $(\alpha_{SA}, \gamma_{SA})$ form a Galois connection.

$$(\mathcal{SA}_G, \subseteq_{SA}) \underset{\alpha_{SA}}{\overset{\gamma_{SA}}{\leftrightarrows}} (\widehat{\mathcal{SA}_G}, \sqsubseteq_{SA})$$

$$\alpha_{SA}(X) = min\{\vec{d} \mid C\vec{x} \leq \vec{d}, \vec{x} \in X\}, \text{ where } min \text{ is applied component-wise.}$$

$$\gamma_{SA}(\vec{d}) \triangleq \{\vec{x} \mid C\vec{x} \leq \vec{d}, \vec{x} \in X\} \qquad \gamma_{SA}(\bot) = \emptyset$$

Figure 6.13 shows the abstract static assignment lattice over an Interval abstract domain. The elements marked in bold corresponds to concrete assignments to SSA variables that maps to a concrete program trace via the program transformation steps, $\mathcal{T}_1'$, $\mathcal{T}_2'$. For convenience, we represent the elements of $\widehat{\mathcal{SA}_G}$ using the standard "box" notation to represent Intervals in Figure 6.13.

Definition 6.8.9. (Meet Irreducibles of $\widehat{\mathcal{SA}_G}$)

A meet irreducible of *abstract static assignment* domain $\widehat{\mathcal{SA}_G}$ where C_i is i-th row vector of a matrix of size $N \times M$ and $\vec{d}$ is a vector of size N, is defined below. Here, MAX is the largest value of $\vec{d}$ determined by the matrix C.

$$\sqcap_{irrd}(\widehat{\mathcal{SA}_G}) \triangleq \{\vec{d} \mid \exists_{=1}\, d_i \in \vec{d}.\ (d_i \neq MAX) \wedge C_i\vec{x} \leq d_i\} \text{ where } (i \leq N)$$

Informally, a meet irreducible of $\widehat{\mathcal{SA}_G}$ is the abstract value $\vec{d}$ such that there exists exactly one element of $\vec{d}$ that is not MAX.

We now discuss how MAX is computed. For an element of $\widehat{\mathcal{SA}_G}$, $\begin{pmatrix} 1 & -1 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} \leq d$ where x and y are 32-bit signed integers (marked as s32 in Figure 6.14), the total number of

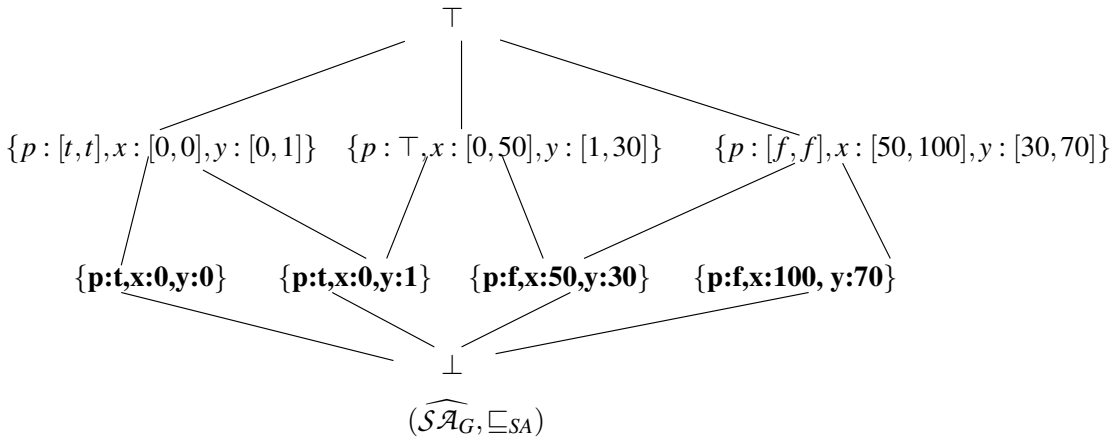


Figure 6.13: Example of Abstract Static Assignment Lattice over a boolean SSA variable p and two numerical SSA variables x and y that take values in the Interval domain

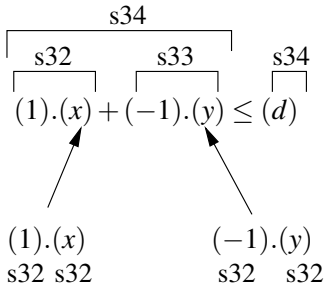


Figure 6.14: Computing MAX from bit-width of d

bits required to represent d is 34. Thus, the value of MAX is the largest value representable in 34 bits.

Meet irreducibles of $\widehat{\mathcal{SA}_G}$ over an *Interval* domain with template $[l,u]$ such that $\begin{pmatrix}-1\\1\end{pmatrix}(x) \leq \begin{pmatrix}-l\\u\end{pmatrix}$ is given by $\{\begin{pmatrix}-l\\MAX\end{pmatrix}, \begin{pmatrix}MAX\\u\end{pmatrix}\}$.

Example 6.8.3. For example, consider an element of $\mathcal{SA}_G$, $p = \{x_1 \geq 2, x_1 - z \leq 30\}$, then $\vec{x} = \begin{pmatrix}x_1\\z\end{pmatrix}$ and abstract value is given by the element,

$$\alpha_{SA}(p) = \begin{pmatrix}MAX\\MAX\\MAX\\-2\\MAX\\30\\MAX\\MAX\end{pmatrix} \text{ with } C = \begin{bmatrix}0 & 1\\1 & 0\\0 & -1\\-1 & 0\\1 & 1\\1 & -1\\-1 & 1\\-1 & -1\end{bmatrix}$$

Example 6.8.4. An example of meet irreducible in $\widehat{\mathcal{SA}_G}$ over $\vec{x} = \begin{pmatrix}a\\b\end{pmatrix}$ that takes values in

Interval domain, for an element of $\mathcal{SA}_G$, $p = \{a \geq 5, a \leq 7\}$, is given by, $\sqcap_{irrd}(\alpha_{SA}(p)) = \vec{d} = \{ \begin{pmatrix} MAX \\ -5 \\ MAX \\ MAX \end{pmatrix}, \begin{pmatrix} MAX \\ MAX \\ MAX \\ 7 \end{pmatrix} \}$ with $C = \begin{pmatrix} 0 & -1 \\ -1 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix}$. Intuitively, this means that $\gamma_{SA}(\vec{d})$ gives the values of a and b that satisfies $a \geq 5 \wedge a \leq 7$.

Example 6.8.5. An example of meet irreducible in $\widehat{\mathcal{SA}_G}$ over $\vec{x} = \begin{pmatrix} a \\ b \end{pmatrix}$ that takes values in the *Octagon* domain, for an element of $\mathcal{SA}_G$, $\{a+b \leq 5\}$, is given by,

$$\sqcap_{irrd}(\alpha_{SA}(\{a+b \leq 5\})) = \begin{pmatrix} MAX \\ MAX \\ MAX \\ MAX \\ 5 \\ MAX \\ MAX \\ MAX \end{pmatrix}$$

$$\text{with } C = \begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 0 & -1 \\ -1 & 0 \\ 1 & 1 \\ 1 & -1 \\ -1 & 1 \\ -1 & -1 \end{bmatrix}$$

Proposition 6.8.1. $(\mathbb{P}(\Pi), \subseteq) \xleftarrow{\gamma_T} (\mathcal{SA}_G, \subseteq_{SA}) \underset{\alpha_{SA}}{\overset{\gamma_{SA}}{\leftrightarrows}} (\widehat{\mathcal{SA}_G}, \sqsubseteq_{SA})$

Proof. The lattice $\mathcal{SA}_G$ is obtained from the concrete lattice of traces $\mathbb{P}(\Pi)$ via syntactic translation steps $\mathcal{T}_1$ and $\mathcal{T}_2$, as shown in Figure 6.1. Both translation steps are exact, that is, $\mathbb{P}(\Pi)$ and $\mathcal{SA}_G$ form an exact abstraction via the concretization-based Galois connection γ_T. This implies that for $\tau \in \mathcal{SA}_G$, $\gamma_T(\tau) = \pi$, where $\pi \in \mathbb{P}(\Pi)$, since an SSA can be exactly mapped back to the program trace following the syntactic translation steps proposed in [39]. Thus, there is no loss or gain of information between elements of $\mathcal{SA}_G$ and $\mathbb{P}(\Pi)$.

Furthermore, the lattice $\widehat{\mathcal{SA}_G}$ is obtained from $\mathcal{SA}_G$ via the abstraction function α_{SA} where α_{SA} is an overapproxation by Definition 6.8.8. Also, $\mathcal{SA}_G$ is an exact abstraction of $\mathbb{P}(\Pi)$. By transitivity, the lattice $\widehat{\mathcal{SA}_G}$ overapproximates $\mathbb{P}(\Pi)$. □

The abstract transformers for the lattice $\widehat{\mathcal{SA}_G}$ transform a memory state of a program and therefore are associated with state transformers. For every constraint $s \in \Sigma^\dagger$, let $apost_s$ and $apre_s$ be sound overapproximations of $post_s$ and pre_s in the lattice $\widehat{\mathcal{SA}_G} \to \widehat{\mathcal{SA}_G}$,

respectively. The global abstract static assignment transformers for the lattice $\widehat{\mathcal{SA}_G}$ over a set of constraints $\Sigma^\dagger$ are obtained from abstract state transformers, $apost_s, apre_s$. This is defined next.

Definition 6.8.10. (Global Overapproximate Abstract Static Assignment Transformers).

$$apost_{\Sigma^\dagger}, apre_{\Sigma^\dagger} : \widehat{\mathcal{SA}_G} \to \widehat{\mathcal{SA}_G}$$

$$apost_{\Sigma^\dagger}(a) \hat{=} \prod_{\sigma \in \Sigma^\dagger} apost_\sigma \circ a$$

$$apre_{\Sigma^\dagger}(a) \hat{=} \prod_{\sigma \in \Sigma^\dagger} apre_\sigma \circ a$$

The transformers $apost_{\Sigma^\dagger}$ and $apre_{\Sigma^\dagger}$ soundly overapproximates $post_{\Sigma^\dagger}$ and $pre_{\Sigma^\dagger}$, respectively.

6.8.4 Complementation in Abstract Static Assignment Lattice

Recall that every meet irreducible of a partial assignments domain admits a precise complement. This property of the partial assignments domain enables a CDCL solver to learn a conflict clause that is obtained by complementing a conflict reason. In Section 6.8.2, we showed that the elements of abstract flow lattice $\widehat{\mathcal{F}_G}$ do not always have precise complements. In this section, we show that the meet irreducibles of $\widehat{\mathcal{SA}_G}$ have precise complements.

Definition 6.8.11. A *meet decomposition* $decomp(a)$ of an abstract element $a \in \widehat{\mathcal{SA}_G}$ is a set of meet irreducibles $M \subseteq \widehat{\mathcal{SA}_G}$ such that $a = \prod_{m \in M} m$.

Example 6.8.6. The meet decomposition of the interval domain element $decomp(2 \leq x \leq 4 \wedge 3 \leq y \leq 5)$ is the set $\{x \geq 2, x \leq 4, y \geq 3, y \leq 5\}$.

Recall that a precise complement of an element $a \in \widehat{\mathcal{SA}_G}$ is an element $\bar{a} \in \widehat{\mathcal{SA}_G}$ such that $\neg\gamma_{SA}(\bar{a}) = \gamma_{SA}(a)$. The meet irreducibles of Abstract Static Assignment lattice $\widehat{\mathcal{SA}_G}$ admits precise complements. This is explained below.

$$\gamma_{SA}(\vec{d}) \hat{=} \{\vec{x} \mid C\vec{x} \leq \vec{d}\}$$

$$\gamma_{SA}(\bar{\vec{d}}) \hat{=} \{\vec{x} \mid C\vec{x} > \vec{d}\}$$

$$\neg\gamma_{SA}(\bar{\vec{d}}) \hat{=} \{\vec{x} \mid C\vec{x} \leq \vec{d}\}$$

Example 6.8.7. Let us consider an element $a \in \widehat{\mathcal{SA}_G}$ such that $decomp(a) = \{p : t, x \geq 0, y \geq 0\}$, then $\bar{a} = \{p : f, x < 0, y < 0\}$ and $\neg\bar{a} = \{p : t, x \geq 0, y \geq 0\}$.

Domain Name	Concrete Domain	Abstract Elements	Meet Irreducible	Complemented Meet Irreducible
Interval	$\mathbb{P}(Var \to \mathbb{Z}) \cup \bot$	$x = [l,u]$	$x \leq N$	$x > N$
Octagon	$\mathbb{P}(Var \times Var \to (\mathbb{Z} \cup \{\infty\})) \cup \bot$	$(\pm x_i \pm x_j \leq d)$	$(x+y \leq N)$	$(-x-y < N)$
Equality	$\mathbb{P}(Var \times Var) \cup \bot$	$(x = y)$	$(x = y)$	$(x \neq y)$
Zones	$\mathbb{P}(Var \times Var \to (\mathbb{Z} \cup \{\infty\})) \cup \bot$	$(x_i - x_j \leq d)$	$(x+y < N)$	$(-x-y \leq N)$
Polyhedra	$\mathbb{P}(Var \times Var \to (\mathbb{Z} \cup \{\infty\})) \cup \bot$	$(a_1 x_i + \ldots + a_n x_n \leq d)$	$(2x+4y < N)$	$(-2x-4y \leq N)$

Table 6.1: Instances of template-based abstract domain

Example 6.8.8. The precise complement of a meet irreducible $(x \leq 2)$ in the interval domain over integers is $(x \geq 3)$, or the precise complement of the meet irreducible $(x+y \leq 1)$ in the octagon domain over integers is $(x+y \geq 2)$.

Table 6.1 shows that most abstract domains admit precise complements. However, complementation of domain elements are not commonly used in classical abstract interpretation. We now show that the meet irreducibles of $\widehat{\mathcal{SA}_G}$ have precise complements when instantiated over these abstract domains.

Proposition 6.8.2. $\gamma_{SA}(C_i\vec{x} \leq \vec{d}_i) = (\neg\gamma_{SA}(C_i\vec{x} > \vec{d}_i))$, where $\vec{d}$ is a vector of size N and $1 \leq i \leq N$.

Assuming that $\vec{d}$ is chosen from a non-relational domain such as Interval domain *Itvs*, the meet irreducibles are of the form $\{\vec{x} \diamond \vec{d}\}$, where $\diamond = \{<, >, \leq, \geq\}$. It is easy to see that the meet irreducibles of *Itvs* are easily complementable. Similarly, assuming that $\vec{d}$ is chosen from a relational domain such as Octagon domain *Octs*, the meet irreducibles are of the form $\{C\vec{x} \diamond \vec{d}\}$, which also admits precise complements. However, for element $a \in \widehat{\mathcal{SA}_G}$, that does not admit precise complementation, it can be decomposed into half spaces, $decomp(a) = \{m_1, m_2, \ldots m_k\}$, such that each m_i admits precise complements.

6.9 Abstract Conflict Driven Learning for Safety (ACDLS)

Figure 6.15 present our framework called *Abstract Conflict Driven Learning for Safety* that uses abstract model search and abstract conflict analysis procedures for sound and precise safety verification. The model search procedure operates on an overapproximate abstract domain using sound deduction transformers such as strongest postcondition or existential precondition transformers (see Section 6.9.1). When the deduction transformers cannot infer any further information and do not meet a γ-completeness condition (see Definition 6.3.2), then a decision is made that refines the current abstract element. The decision and deduction steps continue until either a satisfying assignment is obtained (i.e., the corresponding deduction transformer is γ-complete) or a conflict is encountered. In the

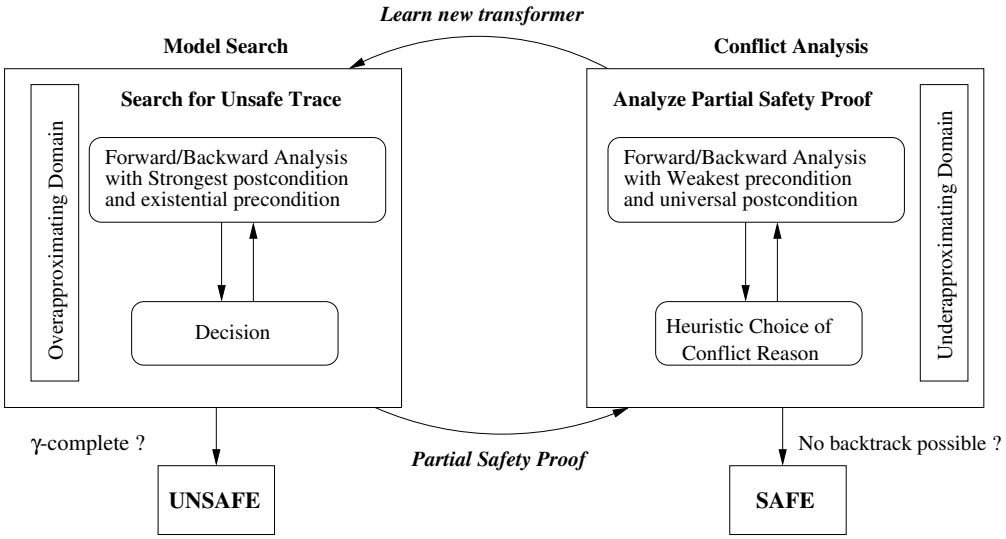


Figure 6.15: ACDLS: Abstract Conflict Driven Learning for Safety

former case, ACDLS terminates with a counterexample trace and the program is *unsafe*. Recall that a counterexample trace is a trace that reaches the error location Ξ. However, if a conflict is encountered, then it implies that the corresponding program trace is either not valid or safe. ACDLS then moves to the conflict analysis phase (see Section 6.9.2) where it learns the reason for the conflict. Recall that a SAT solver uses conflict resolution to derive the reason for the conflict. Conflict analysis in ACDLS operates on an underapproximate domain using sound abductive transformers such as universal postcondition or weakest precondition transformers (see Section 6.9.2). There can be multiple incomparable reasons for the conflict, but ACDLS heuristically picks one reason and generalizes it. Intuitively, this means that a partial safety proof for safe or invalid trace $\mathcal{S}$ is obtained by generalizing $\mathcal{S}$ to a set of safe or invalid traces $\mathcal{S}'$ such that the generalized conflict reason still preserves the reachability of the conflict.

Recall from Section 6.7.2 that the SSA is represented by a set of constraints $\Sigma^\dagger$. The learned transformer is obtained by complementing the conflict reason and added to $\Sigma^\dagger$. An invariant of the ACDLS algorithm is that the set of transformers after learning preserves the reachability of the error location Ξ. After learning, ACDLS backtracks non-chronologically and undoes all deductions until the decision level where the analysis is non-conflicting. Model search is repeated again with the new learned transformer. However, if no further backtracking is possible, then ACDLS terminates and returns *safe*.

In the subsequent sections, we present a new abstract interpretation framework for generalizing CDCL to sound and precise safety verification over the template-based relational abstract domains.

6.9.1 Abstract Model Search

The model search in a CDCL solver alternates between two phases – *decisions* and *Boolean Constraint Propagation*, until a satisfying assignment is obtained or a conflict is encountered. Silva et al. [89] shows that BCP computes a greatest fixed point by applying a unit rule which is the best abstract transformer over a partial assignments domain. Here, we characterize abstract model search as a procedure to find unsafe traces in programs. To do so, we present abstract model search as an instance of the *Global Bottom Problem*, which is described in Algorithm 3. We now present the deduction and decision transformers which are required in Algorithm 3 for computing the fixed point of the concrete unsafe trace transformer.

Abstract Deduction Transformer

Recall from Section 6.6 that f^G_{unsafe} is a lower closure operator, which may be approximated by computing a greatest fixed point in the abstract domain. We now formally define a transformer, $f^G_{aunsafe}$, over $\widehat{\mathcal{SA}_G}$, that uses strongest postcondition and existential precondition to perform forward, backward and multi-way analysis. This transformer soundly approximates f^G_{unsafe}. We will call it *abstract deduction transformer*.

Definition 6.9.1. (Unsafe Trace Transformer in $\widehat{\mathcal{SA}}$). $f^{\rightarrow}_{aunsafe}, f^{\leftarrow}_{aunsafe} : \widehat{\mathcal{SA}_G} \rightarrow \widehat{\mathcal{SA}_G}$

$$f^{\rightarrow}_{aunsafe}(A) \hat{=} gfp\ Z.\ apost_{\Sigma^\dagger}(A \sqcap Z) \quad f^{\leftarrow}_{aunsafe}(A) \hat{=} gfp\ Z.\ apre_{\Sigma^\dagger}(A \sqcap Z)$$

$$f^G_{aunsafe}(A) = f^{\leftarrow}_{aunsafe}(A) \sqcap f^{\rightarrow}_{aunsafe}(A)$$

Proposition 6.9.1. The transformer $f^G_{aunsafe}$ soundly approximate f^G_{unsafe}.

Proof. We know that $f^G_{aunsafe} = f^{\leftarrow}_{aunsafe} \sqcap f^{\rightarrow}_{aunsafe}$. We prove the case that $f^{\rightarrow}_{aunsafe}$ overapproximates f^G_{unsafe}. The proof for $f^{\leftarrow}_{aunsafe}$ is similar.

Recall from Section 6.7 that SA_G is an exact abstraction of $\mathbb{P}(\Pi)$ which is obtained via the syntactic translation steps, $\mathcal{T}_1$ and $\mathcal{T}_2$. The lattice $\widehat{\mathcal{SA}_G}$ soundly overapproximates $\mathbb{P}(\Pi)$ following Proposition 6.8.1. Also, the global static assignment transformer $apost_{\Sigma^\dagger}$ soundly approximate $tpost_G$. Thus, for any element $p \in \widehat{\mathcal{SA}_G}$, the abstract transformer $Z \mapsto apost_{\Sigma^\dagger}(p \sqcap_{SA} Z)$ soundly overapproximate the concrete transformer which is given by $Z \mapsto \gamma_T(\gamma_{SA}(p)) \cap (I \cup tpost_G(Z))$. From the fixed point transfer theorem [69], it follows that $\mathsf{gfp}\ Z.apost_{\Sigma^\dagger}(p \sqcap_{SA} Z)$ soundly approximates $\mathsf{lfp}\ Z.\gamma_T(\gamma_{SA}(p)) \cap (I \cup tpost_G(Z))$. Hence, we proved that $f^{\rightarrow}_{aunsafe}$ overapproximates f^G_{unsafe}. Therefore, a trace $\pi \in \gamma_T \circ \gamma_{SA} \circ f^G_{aunsafe}$ implies that $\pi \in f^G_{unsafe} \circ \gamma_T \circ \gamma_{SA}$. □

Generalized Decision Operator

An abstract model search heuristically searches for counterexample trace or a conflict. This process generates a downward iteration sequence in the lattice of $\widehat{\mathcal{SA}_G}$. A decision in a CDCL solver heuristically picks an unassigned variable and assigns a value to it. Similarly, a decision in ACDLS refines a downwards iteration sequence by jumping under the greatest fixed point when the transformers $apost_s$ and $apre_s$ fail to make a refinement. Given an abstract element (a) obtained from the fixed point iteration, a decision f_{dec} in $\widehat{\mathcal{SA}_G}$ heuristically chooses a meet irreducible (a'), such that the resultant element $a \sqcap a'$ is strictly smaller than (a). We formally define a generalized decision operator over $\widehat{\mathcal{SA}_G}$.

Definition 6.9.2. (Generalized Decision Operator) $f_{dec}\colon \widehat{\mathcal{SA}_G} \times \widehat{\mathcal{SA}_G} \to \widehat{\mathcal{SA}_G}$

$$f_{dec}(a,a') = a \sqcap a', \text{such that } a \neq a' \wedge (a \sqcap a' \sqsubseteq a) \wedge (a \sqcap a' \sqsubseteq a')$$

Global Bottom Problem

We now define the Global Bottom Problem. We show that program correctness or reachability is an instance of global bottom problem. Given a lattice of program traces, and an unsafe trace transformer $f^G_{aunsafe}$ that returns set of unsafe and well-formed traces, if the application of $f^G_{aunsafe}$ returns $\bot$ for every element in the lattice of program traces, then the program is *SAFE*.

Definition 6.9.3. Given an abstract static assignment lattice, $(\widehat{\mathcal{SA}_G}, \sqsubseteq_{SA})$, a transformer $f^G_{aunsafe} : \widehat{\mathcal{SA}_G} \to \widehat{\mathcal{SA}_G}$ is *globally bottom* if $f^G_{aunsafe}(a) = \bot$ for all $a \in \widehat{\mathcal{SA}_G}$. The global bottom problem is to determine if a transformer $f^G_{aunsafe}$ on a lattice $\widehat{\mathcal{SA}_G}$ is globally bottom.

An element $a \in \widehat{\mathcal{SA}_G}$ is a non-$\bot$ witness if $f^G_{aunsafe}(a) \neq \bot$. Program correctness can be seen as an instance of the Global Bottom Problem since a non-$\bot$ witness corresponding to the abstract state in the error location Ξ of an well-formed trace implies that the program is unsafe.

We use an alternative characterization of f^G_{unsafe} transformer for instantiating the Global Bottom Problem described above. Recall from Proposition 6.6.3 that the transformer f^G_{unsafe} is completely additive and reductive. In this dissertation, we consider the global bottom problem for a completely additive and reductive transformer f^G_{unsafe} on powerset lattices, $(\mathbb{P}(\Pi), \subseteq)$. The result in Proposition 6.9.2 follows directly from the fixed-point tranfer theorem [70].

Proposition 6.9.2. Given a completely additive and reductive transformer f^G_{unsafe} on powerset of traces $(\mathbb{P}(\Pi), \subseteq)$, let $f^G_{aunsafe}\colon \widehat{\mathcal{SA}_G} \to \widehat{\mathcal{SA}_G}$ be a sound overapproximation of f^G_{unsafe}, and $\gamma_{SA}(gfp(f^G_{aunsafe})) = \bot$, then the transformer f^G_{unsafe} is globally bottom.

We now provide a condition to check whether f^G_{unsafe} is not globally bottom. For this, it is sufficient to check whether $f^G_{aunsafe}$ is γ-complete even though the underlying abstract domain and the transformer may still be imprecise.

Proposition 6.9.3. If $f^G_{aunsafe}$ is γ-complete at an element $a \in \widehat{\mathcal{SA}_G}$ and $\gamma_{SA}(f^G_{aunsafe}(a)) \neq \bot$, then f^G_{unsafe} is not globally bottom.

Algorithm 3 presents a procedure for computing a non-$\bot$ witness to the Global Bottom Problem. The algorithm takes as input an approximate abstract deduction transformer, $f^G_{aunsafe}$, a generalized decision transformer, f_{dec}, and a trail data structure $\mathcal{T}$ implemented as a stack that record the results of transformer applications, that is, $\mathcal{T}$ contains elements of $\widehat{\mathcal{SA}_G}$. $\mathcal{T}$ is initialized to $\top$. The expression $\bigsqcap \mathcal{T}$ denotes conjunction of all elements in $\mathcal{T}$. The expression $\mathcal{T} \leftarrow \mathcal{T}.a$ denotes concatenating $\mathcal{T}$ with the new element a which is pushed in $\mathcal{T}$. For $a = \bigsqcap \mathcal{T}$, the algorithm checks if $\forall\, d \sqsubset_{SA} a. f^G_{aunsafe}(d) = \bot$, that is, $f^G_{aunsafe}$ is bottom for every elements below $a \in \widehat{\mathcal{SA}_G}$ (initially, $a = \top$). The output of the algorithm is a tuple consisting of the result of abstract deduction transformer which can be $\{\bot$, non-$\bot$, UNKNOWN$\}$ and the final content of $\mathcal{T}$. Following Proposition 6.9.2, if $\gamma_{SA}(\mathsf{gfp}(f^G_{aunsafe}(a))) = \bot$, then $\forall\, c \sqsubset_{SA} \gamma_{SA}(a). f^G_{unsafe}(c) = \bot$, that is, f^G_{unsafe} is bottom on every elements c that are below $\gamma_{SA}(a)$. Note that, if the initial value of $a = \top$, and `abstract-counterexample-search` returns $\bot$, that is, $\gamma_{SA}(gfp(f^G_{aunsafe}(\top))) = \bot$, then f^G_{unsafe} is *globally bottom*. Else, for $a \neq \top$, $\gamma_{SA}(\mathsf{gfp}(f^G_{aunsafe}(a))) = \bot$ corresponds to a partial safety proof. However, if $f^G_{aunsafe}$ is γ-complete at a fixed point $a^\dagger$, and $\gamma_{SA}(f^G_{aunsafe}(a^\dagger)) \neq \bot$, then following Proposition 6.9.3, f^G_{unsafe} is *not globally bottom.* If none of these conditions holds true, then a generalized decision operator f_{dec} is applied, which jumps under the greatest fixed point and checks if $f^G_{aunsafe}$ is $\bot$ on elements below the fixed point.

Example 6.9.1. Figure 6.17 shows an example of the abstract model search procedure for the CFG of Figure 6.16 over the Interval domain. The abstract values computed by the transformer $f^G_{aunsafe}$ with strongest postcondition *apost* are marked in blue in Figure 6.17. Starting from the initial value $\top$, forward analysis concludes that x is between -2 and 2 by the application of the transformer $apost_{x:=-2} \cup apost_{x:=0} \cup apost_{x:=2}$. Note that the loop in the CFG of Figure 6.16 between the nodes $n6$ and $n7$ is completely unwound and all the statements associated with this loop are collectively referred to as $loop$ in Figure 6.17. A forward fixed point analysis of the $loop$ (marked by $apost_{loop}$) does not yield any new information. Clearly, the analysis is imprecise to conclude anything about the reachability of the error location $Error$. Hence, we apply a decision by picking a meet irreducible $y \geq 2$. We then apply forward analysis from this decision which yields a downward iteration

Algorithm 3: Abstract search for a non-$\bot$ witness of Global Bottom Problem abstract-counterexample-search($f^G_{aunsafe}, f_{dec}, \mathcal{T}$)

input : $f^G_{aunsafe}$: $\widehat{\mathcal{SA}_G} \to \widehat{\mathcal{SA}_G}$, f_{dec}: $\widehat{\mathcal{SA}_G} \times \widehat{\mathcal{SA}_G} \to \widehat{\mathcal{SA}_G}$, $\mathcal{T}$: elements of $\widehat{\mathcal{SA}_G}$
output : A tuple $(result, \mathcal{T})$, where result can be $\bot$ or non-$\bot$

1 $a \leftarrow \bigsqcap \mathcal{T}$
2 **do**
3 $\quad a^\dagger \leftarrow a$
4 $\quad a \leftarrow a \sqcap_{SA} f^G_{aunsafe}(a)$
5 **while** $a = a^\dagger$ *or* $\gamma_{SA}(a) = \bot$
6 **if** $\gamma_{SA}(a) = \bot$ **then return** $(\bot, \mathcal{T})$
7 $\mathcal{T} \leftarrow \mathcal{T}.a$
8 **if** $f^G_{aunsafe}$ *is* $\gamma-$*complete at* a **then return** (non-$\bot, \mathcal{T}$)
9 $d \leftarrow f_{dec}$
10 $\mathcal{T} \leftarrow \mathcal{T}.d$
11 **return** abstract-counterexample-search($f^G_{aunsafe}, f_{dec}, \mathcal{T}$)

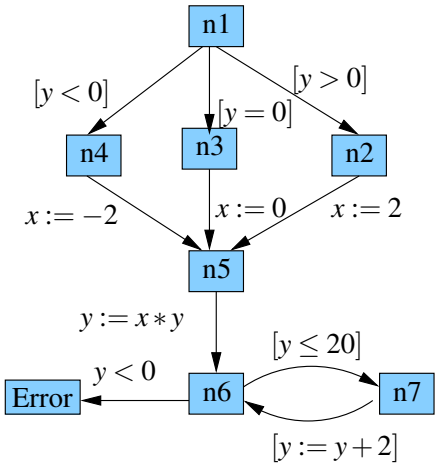


Figure 6.16: An example CFG

sequence as shown in lower part of Figure 6.17. Forward analysis computes the abstract value $\{y \geq 4, x \geq 2, x \leq 2\}$ from the transformer $apost_{y:=x*y}$ and $apost_{loop}$. The application of the transformer $apost_{(y<0)}(y \geq 4 \wedge x \geq 2 \wedge x \leq 2)$ leads to a *conflict*, which is marked by $\bot$ in Figure 6.17. Hence, the error location $Error$ is unreachable for the decision $y \geq 2$.

6.9.2 Abstract Conflict Analysis

The conflict analysis procedure in a CDCL solver finds the reason for a conflict by analyzing the deductions made during the model search phase through *conflict resolution* [21]. Conflict analysis in a CDCL solver is different from conflict analysis in a DPLL solver – CDCL allows non-chronological backjumping which can discard multiple levels of decisions and

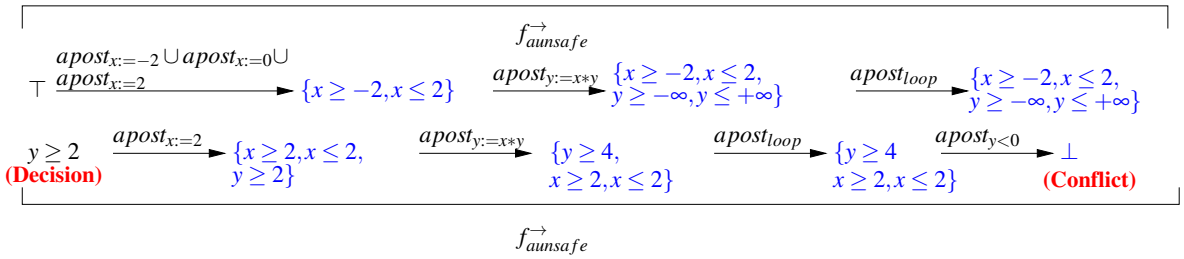


Figure 6.17: Abstract model search with decisions and deductions

deductions in the trail. CDCL learns a reason for a conflict called *conflict clause* that prevents the model search from re-entering into the conflicting search space in the future. A conflict clause in a CDCL solver is a clause that expresses the fact that some combinations of variable assignment are not valid. Haller et al. in [89] shows that conflict analysis in a CDCL solver operates over an underapproximate domain, which is a downward closed set of partial assignments.

Conflict analysis in ACDLS solver can be seen as abductive reasoning [86, 161]. Given a statement S, the logical abduction is the process of extracting from S a reason R whose truth suffices to ensure the truth of S with respect to some background theory. The application of the abductive transformer, say $abd_S(A)$, can be viewed as finding the weakest semantically expressed reason for A assuming the truth of S. However, the notion of abduction in this work formalizes operations for finding the reason for conflict. These operations finds logical statements that are sufficient for deriving a conflict. Logical abduction does not cover such operation and requires the result of abduction to be consistent with the background theory. However, the conflict reason does not satisfy such requirement.

Hence, abductive inference in ACDLS is the dual of deductive inference that takes an input set A contradicting a φ and finds a weakest reason or explanation for A, assuming that φ is unsatisfiable. The program analysis counterpart of abductive inference is obtained through application of the weakest precondition transformer.

A safe trace transformer, f^G_{safe}, computes a set of safe or invalid traces. Following Proposition 6.6.5, the transformer f^G_{safe} is completely multiplicative and extensive. The conflict analysis for safety verification requires a set of transformers that underapproximates f^G_{safe} over the downset lattice of $\widehat{\mathcal{SA}_G}$. The conflict analysis is an operation on sets of conflict reasons, hence the downset completion domain is considered due to the following properties – 1) Downset completion enriches the abstract domain with disjunctions, 2) Downset completion domain underapproximate the concerete powerset lattice [71], 3) Downsets are downwards closed, so any element that refines the conflict reason also belongs to the downset.

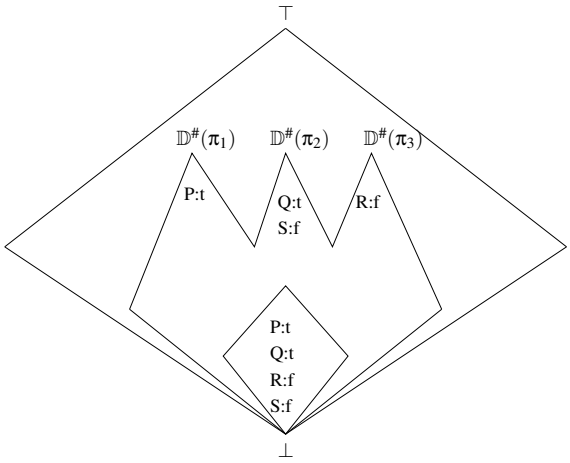


Figure 6.18: Conflict reasons as set of Downsets

CDCL-based SAT solvers uses conflict minimization [202] technique for deriving the conflict reason. Given a partial assignment π that leads to a conflict, conflict minimization replace π with π' such that π can be derived from π' following unit rule. Note that, there can be multiple incomparable partial assignments that lead to the conflict, so downsets of partial assignments have to be considered. The sets of downsets obtained from a partial assignment $\pi = \{P : t, Q : t, R : f, S : f\}$ which lead to a conflict, is presented in Figure 6.18. Given a lattice with $\top$ and $\bot$, the innermost diamond represents the partial assignment π. Let us assume that each partial assignment, $\pi_1 = \{P : t\}$, $\pi_2 = \{Q : t, R : f\}$ and $\pi_3 = \{S : f\}$, obtained from π, is sufficient to derive the conflict. Now, each $\pi_i.i \in \{1,2,3\}$, forms a downset, denoted by $\mathbb{D}^{\#}(\pi_i)$, which is shown by the respective cone-shaped objects in Figure 6.18. A downset completion lattice contains sets of downsets with some ordering, denoted by $\mathbb{D}$ (see Definition 6.3.4). Further, π_1, π_2 and π_3 are incomparable, as is evident from their position in the lattice (peak of respective cones). A point to note here is that if π_i leads to a conflict, then every element that refines π_i must belong to the downset of π_i and must also contribute to the conflict.

Recall that a program transformer $s \in \Sigma$ transforms the memory state of a program and is associated with a transition relation, $\mathcal{ST}_s^{\Omega}$. We can associate each $s \in \Sigma$ with a weakest pre-condition $\widehat{pre_s}$ and a universal postcondition $\widehat{post_s}$. An abstract conflict analysis procedure computes an underapproximation of the concrete safe trace transformer f_{safe}^G. To compute an underapproximation of f_{safe}^G, we underapproximate the state transformers, $\widehat{post_s}$ and $\widehat{pre_s}$. Furthermore, the lattice $\mathcal{SA}_G$ is underapproximated by the downset completion $\mathbb{D}(\widehat{\mathcal{SA}_G})$, which is the set of downsets of $\mathcal{SA}_G$, as shown in Figure 6.19. In this dissertation, we treat downset lattice as underapproximating abstractions since $\mathbb{D}(\widehat{\mathcal{SA}_G})$ underapproximates the concrete lattice $\mathcal{SA}_G$.

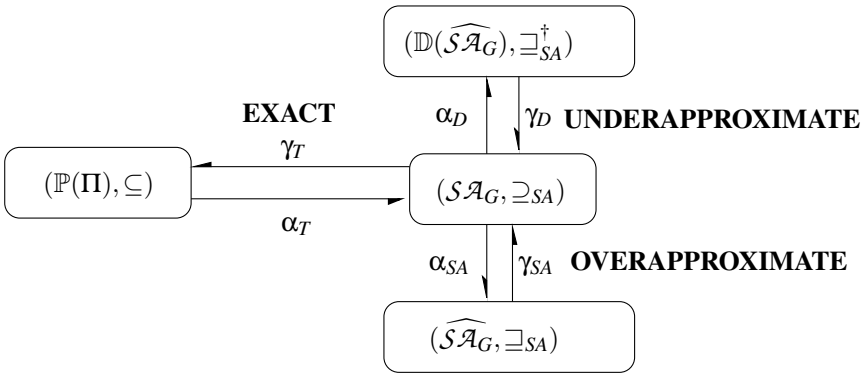


Figure 6.19: Lattice for Conflict Analysis

Proposition 6.9.4.

$$(\mathcal{SA}_G, \supseteq_{SA}) \underset{\alpha_D}{\overset{\gamma_D}{\leftrightarrows}} (\mathbb{D}(\widehat{\mathcal{SA}_G}), \sqsupseteq^{\dagger}_{SA})$$

$$\alpha_D(a) \hat{=} \{a' \in \mathbb{D}(\widehat{\mathcal{SA}_G}) \mid \gamma_{SA}(a') \subseteq_{SA} a\} \qquad \gamma_D(b) \hat{=} \{ \bigcup_{b' \in decomp(b)} \gamma_{SA}(b') \}$$

Proof. We now prove that the pair (α_D, γ_D) forms a Galois connection. It is straightforward to see that α_D and γ_D are monotone. We show that $(\alpha_D \circ \gamma_D)$ is extensive and $(\gamma_D \circ \alpha_D)$ is reductive.

Let us assume that $a \in \mathbb{D}(\widehat{\mathcal{SA}_G})$. Then, $\gamma_D(a) \supseteq_{SA} \gamma_{SA}(a)$, and thus $a \in \alpha_D \circ \gamma_D$. Thus, $\alpha_D \circ \gamma_D$ is extensive.

Let $c \in \gamma_D \circ \alpha_D(c')$. Then, there exists an element $a \in \alpha_D(c')$ such that $c \in \gamma_{SA}(a)$ and $\gamma_{SA}(a) \subseteq_{SA} c'$ (by Galois connection above). Therefore, $c \in \gamma_D(a)$. Thus, $\gamma_D \circ \alpha_D$ is reductive. □

For effective conflict analysis, we require to associate each overapproximate strongest postcondition and existential precondition with an underapproximate weakest precondition and universal postcondition respectively. The safe trace transformer f^G_{safe} is decomposed into local steps that underapproximate weakest precondition and universal postcondition transformer. Let $\widehat{apre_s}$ and $\widehat{apost_s}$ be sound underapproximations of the weakest precondition transformer $\widehat{pre_s}$ and universal postcondition transformer $\widehat{post_s}$ over $\mathbb{D}(\widehat{\mathcal{SA}_G})$. The global abstract static assignment transformers for the lattice $\mathbb{D}(\widehat{\mathcal{SA}_G})$ are obtained from the underapproximate abstract state transformers, $\widehat{apost_s}$, $\widehat{apre_s}$. This is defined next.

Definition 6.9.4. (Global Underapproximate Abstract Static Assignment Transformers for $\mathbb{D}(\widehat{\mathcal{SA}_G})$).

$$\widehat{apost_{\Sigma^\dagger}}, \widehat{apre_{\Sigma^\dagger}} : \mathbb{D}(\widehat{\mathcal{SA}_G}) \to \mathbb{D}(\widehat{\mathcal{SA}_G})$$

$$\widehat{apost_{\Sigma^\dagger}}(a) \hat{=} \prod_{\sigma \in \Sigma^\dagger} \widehat{apost_\sigma} \circ a$$

$$\widehat{apre_{\Sigma^\dagger}}(a) \mathrel{\hat{=}} \prod_{\sigma \in \Sigma^\dagger} \widehat{apre_\sigma} \circ a$$

The transformers, $\widehat{apost_{\Sigma^\dagger}}$ and $\widehat{apre_{\Sigma^\dagger}}$, soundly underapproximate their concrete counterparts, $\widehat{post_{\Sigma^\dagger}}$ and $\widehat{pre_{\Sigma^\dagger}}$ respectively.

A conflict reason is derived by analyzing the deductions made from $f^G_{aunsafe}$ during the model search phase. Conflict analysis is performed by computing a least fixed point of a weakest precondition transformer or a universal postcondition transformer in the downset abstract domain $\mathbb{D}(\widehat{\mathcal{SA}_G})$. For this, we define a transformer f^G_{asafe} in $\mathbb{D}(\widehat{\mathcal{SA}_G})$, that is a sound underapproximation of the completely multiplicative and extensive transformer f^G_{safe}. Below, A is the downset closure of the original conflict reason.

Definition 6.9.5. (Safe Trace Transformers in $\mathbb{D}(\widehat{\mathcal{SA}})$).

$$f^{\rightarrow}_{asafe}, f^{\leftarrow}_{asafe} : \mathbb{D}(\widehat{\mathcal{SA}_G}) \to \mathbb{D}(\widehat{\mathcal{SA}_G})$$

$$f^{\rightarrow}_{asafe}(A) = lfp\ Z.\ \widehat{apost_{\Sigma^\dagger}}(A \sqcup Z) \quad f^{\leftarrow}_{asafe}(A) = lfp\ Z.\ \widehat{apre_{\Sigma^\dagger}}(A \sqcup Z)$$

$$f^G_{asafe}(A) = f^{\rightarrow}_{asafe}(A) \sqcap f^{\leftarrow}_{asafe}(A)$$

Global Top Problem

We now describe the *Global Top Problem* for the approximation of f^G_{safe} transformer. We show that abstract conflict analysis is an instance of the global top problem.

Definition 6.9.6. Given a downset lattice, $(\mathbb{D}(\widehat{\mathcal{SA}_G}), \sqsupseteq^\dagger_{SA})$, a transformer $f^G_{asafe} : \mathbb{D}(\widehat{\mathcal{SA}_G}) \to \mathbb{D}(\widehat{\mathcal{SA}_G})$ is *globally top* if $f^G_{asafe}(a) = \top$ for all $a \in \mathbb{D}(\widehat{\mathcal{SA}_G})$. The global top problem is to determine if a transformer f^G_{asafe} on a lattice $\mathbb{D}(\widehat{\mathcal{SA}_G})$ is globally top.

An element $a \in \mathbb{D}(\widehat{\mathcal{SA}_G})$ is a non-$\top$ witness if $f^G_{asafe}(a) \neq \top$. Algorithm 4 present a procedure for checking if f^G_{asafe} is globally top. Recall from Section 6.6 that f^G_{safe} is completely multiplicative and extensive. Following Proposition 6.6.4, any multiplicative and extensive function on a powerset lattice is a upper closure. An upper closure operator may be approximated by least fixed point in the abstract.

Assuming that f^G_{asafe} and $f^G_{aunsafe}$ are best abstract transformers, if the result of $\mathsf{lfp}(f^G_{asafe})$ concretizes to $\top$, then we can infer the following.

1. f^G_{safe} is globally top.
2. f^G_{unsafe} is globally bottom since $\mathsf{gfp}(f^G_{aunsafe})$ concretizes to $\bot$.

Algorithm 4: Abstract search for a non-$\top$ witness of Global Top Problem in $\mathbb{D}(\widehat{\mathcal{SA}_G})$, analyze-partial-safety-proof($f^G_{asafe}, int\restriction, u$)

input : $f^G_{asafe}\colon \mathbb{D}(\widehat{\mathcal{SA}_G}) \to \mathbb{D}(\widehat{\mathcal{SA}_G})$, $int\restriction\colon \mathbb{D}(\widehat{\mathcal{SA}_G}) \times \mathbb{D}(\widehat{\mathcal{SA}_G}) \to \mathbb{D}(\widehat{\mathcal{SA}_G})$, $u \in \mathbb{D}(\widehat{\mathcal{SA}_G})$

output : A tuple $(result, t)$, where $result$ can be $\top$ or non-$\top$ and $t \in \mathbb{D}(\widehat{\mathcal{SA}_G})$

1 $a \leftarrow u \sqcup^{\dagger}_{SA} f^G_{asafe}(u)$

2 $t \leftarrow u\ int\restriction\ a$

3 **if** $\gamma_D(t) = \top$ **then return** $(\top, t)$

4 **if** f^G_{asafe} *is* γ*-complete at* t **then return** (non-$\top$, t)

5 **return** analyze-partial-safety-proof($f^G_{asafe}, int\restriction, t$)

The procedure `analyze-partial-safety-proof` of Algorithm 4 is used to find a generalized reason for the conflict. To do so, it computes an underapproximation of the least fixed point of f^G_{safe} transformer using the upwards interpolation, $int\restriction$. Recall that upwards interpolation on a lattice L is a function $int\restriction\colon L \times L \to L$ such that $a \sqsubseteq b \implies a \sqsubseteq int\restriction(a,b) \sqsubseteq b$ for all $a, b \in L$. To avoid considering sets of all generalizations of a conflict, $u\ int\restriction\ a$ (in line 2 of Algorithm 4) is used as a choice function, where u and a are downsets. Line 3 of Algorithm 4 suggests that if the weakest precondition of $\bot$ concretizes to $\top$, then the f_{asafe} is globally top.

Example 6.9.2. Let us revisit the example in Figure 6.16 and the corresponding deductions in Figure 6.17. For the purpose of illustration, we show the deductions obtained in the model search phase in the top flow of Figure 6.20. The bottom flow of Figure 6.20 gives the conflict analysis procedure. Starting from the conflict element $\bot$, we iteratively apply the transformer f^G_{asafe} using the weakest precondition $\widehat{apre}$, the result of which is shown in bold in Figure 6.20. For example, $\widehat{apre_{y<0}}(\bot) = \{y \geq 0\}$. The corresponding abstract value obtained from the transformer $f^G_{aunsafe}$ via the strongest postcondition is $\{y \geq 4\}$. We heuristically pick a generalized element A_0 such that $\{y \geq 4\} \sqsubseteq A_0 \sqsubseteq \{y \geq 0\}$; we pick $A_0 = \{y \geq 0\}$ (marked in blue) using upwards interpolation (See Definition 6.3.6), $int\restriction(y \geq 0, y \geq 4)$. The heuristic generalization of the conflict reason for different abstract domains is explained in Section 7.7.2. Recall that the loop in the CFG of Figure 6.16 between the nodes $n6$ and $n7$ is completely unwound and all the statements associated with this loop are collectively referred to as $loop$ in Figure 6.20. We then apply the transformer $\widehat{apre_{loop}}$ to compute the weakest precondition of the loop. Subsequently, the application of transformers $\widehat{apre_{y:=x*y}}$ and $\widehat{apre_{x:=2}}$ derives a generalized reason, $\{x > 0, y \geq 0\}$, where $y \geq 0$ strictly generalizes the decision $(y \geq 2)$. It is important to note that the partial safety proof corresponding to the decision $y \geq 2$ gives a generalized conflict reason $y \geq 0$, therefore avoiding the decision $y \geq 1$. Taking a trace-based view, the decision $y \geq 2$ corresponds to the

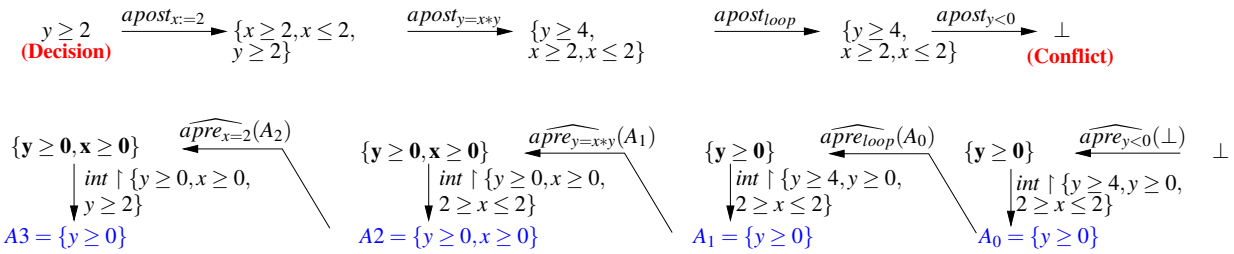


Figure 6.20: Conflict analysis with underapproximate weakest precondition and upwards interpolation

set of traces through the nodes $\{n1 \rightarrow n2 \rightarrow n5 \ldots\}$ in the CFG of Figure 6.16. However, the conflict reason $y \geq 0$ corresponds to the set of traces through the nodes $\{n1 \rightarrow n2 \rightarrow n5 \ldots\}$ as well as set of traces that pass through $\{n1 \rightarrow n3 \rightarrow n5 \ldots\}$ in the CFG of Figure 6.16. Hence, this example demonstrates that the conflict analysis for a partial safety proof can be generalized to the set of safe traces.

Analogous to the conflict generalization of the safe traces is the target enlargement [15, 222] technique used in hardware verifciation. In the target enlargement approach, the search for a path to a given target state is performed by enlarging the target through preimage computation to calculate the set of states which may hit the target within k time-steps. Intuitively, the target is now replaced by a wider area. Hence, the chance to reach the target or to prove that the target is unreachable is now higher. In the context of ACDLS, the generalized conflict reason derived from the partial safety proof can be viewed as an "enlargement" of the target conflict state which gives a sufficient reason for the conflict.

6.9.3 Generalized Unit Rule

Learning new abstract transformers $(AUnit)$ from conflicts in ACDLS refines the abstract unsafe trace transformer $f^G_{aunsafe}$. Intuitively, transformer learning through conflict analysis makes $f^G_{aunsafe}$ a closer approximation of the concrete unsafe trace transformer f^G_{unsafe}. The transformer $AUnit$ is a generalization of the propositional unit rule [80]. Recall that φ is a safety formula that is obtained through conjunction of all constraints in $\Sigma^\dagger$. For an abstract lattice $\widehat{\mathcal{SA}_G}$ with complementable meet irreducibles and a set of meet irreducibles $C \subseteq \widehat{\mathcal{SA}_G}$ such that $\bigsqcap C$ does not satisfy φ, $AUnit_C : \widehat{\mathcal{SA}_G} \rightarrow \widehat{\mathcal{SA}_G}$ is formally defined as follows.

$$AUnit_C(a) = \begin{cases} \bot & \text{if } a \sqsubseteq \bigsqcap C \quad (1) \\ \bar{t} & \text{if } t \in C \text{ and } \forall t' \in C \setminus \{t\}.a \sqsubseteq t' \quad (2) \\ \top & \text{otherwise} \quad (3) \end{cases}$$

Rule (1) of $AUnit$ returns $\bot$ since $a \sqsubseteq \bigsqcap C$ is conflicting. Rule (2) of $AUnit$ infers a valid meet irreducible, which implies that C is unit. Rule (3) of $AUnit$ returns $\top$ which implies

Algorithm 5: Abstract Conflict Driven Learning for Safety

input : $f^G_{aunsafe} : \widehat{\mathcal{SA}_G} \to \widehat{\mathcal{SA}_G}$, $f_{dec} : \widehat{\mathcal{SA}_G} \times \widehat{\mathcal{SA}_G} \to \widehat{\mathcal{SA}_G}$, $f^G_{asafe} : \mathbb{D}(\widehat{\mathcal{SA}_G}) \to \mathbb{D}(\widehat{\mathcal{SA}_G})$, $int \uparrow : \mathbb{D}(\widehat{\mathcal{SA}_G}) \times \mathbb{D}(\widehat{\mathcal{SA}_G}) \to \mathbb{D}(\widehat{\mathcal{SA}_G})$

output : $\bot$ if $f^G_{aunsafe}$ is globally bottom, else a counterexample ($\bigsqcap \mathcal{T}$)

$\mathcal{T} \leftarrow \top$
do
  $(d, \mathcal{T}) \leftarrow$ abstract-counterexample-search$(f^G_{aunsafe}, f_{dec}, \mathcal{T})$
  if $d \neq \bot$ **then return** $(d, \bigsqcap \mathcal{T})$
  $a \leftarrow \bigsqcap \mathcal{T}$
  $(d, t) \leftarrow$ analyze-partial-safety-proof$(f^G_{asafe}, int \uparrow, a)$
  $learn \leftarrow AUnit_t$
  $\mathcal{T} \leftarrow$ backjump$(learn, f^G_{aunsafe}, \mathcal{T})$
  $f^G_{aunsafe} \leftarrow f^G_{aunsafe} \sqcap learn$
while $\mathcal{T}$ *is empty*
return $\bot$

that the learned transformer is not *asserting* after backtracking. This would prevent any new deductions from the learned transformer. Progress is then made by decisions.

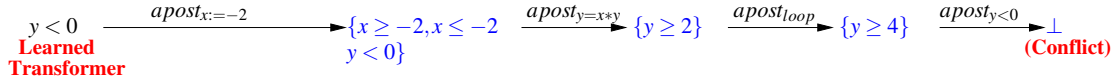


Figure 6.21: Deduction from the learned transformer

Example 6.9.3. Recall from Figure 6.20 that the conflict reason obtained from f^G_{asafe} is $y \geq 0$. The learned transformer is the negation of the conflict reason which is given by $y < 0$. Figure 6.21 shows the fixed point computation with the learned transformer $y < 0$. Taking a trace-based view, the learned transformer $y < 0$ corresponds to the set of traces through the nodes $\{n1 \to n4 \to n5 \ldots\}$ in the CFG of Figure 6.16. Clearly, forward analysis with the learned transformer also leads to the *conflict*. There are no further cases to explore. Thus, ACDLS terminates and returns *safe*.

Algorithm for ACDLS

Algorithm 5 describes the working of Abstract Conflict Driven Learning for Safety. The procedure `abstract-counterexample-search` in ACDLS operates on an overapproximate domain $\widehat{\mathcal{SA}_G}$ with complementable meet irreducibles. The procedure given by `analyze-partial-safety-proof` operates over an underapproximate downset completion domain, $\mathbb{D}(\widehat{\mathcal{SA}_G})$. ACDLS starts by initializing $\mathcal{T}$ to a singleton sequence

$\top$. The outer loop of the algorithm terminates when $\mathcal{T}$ is empty. If the counterexample search procedure `abstract-counterexample-search` returns non-$\bot$, then ACDLS terminates with a counterexample given by $(\bigsqcap \mathcal{T})$. On the other hand, when the procedure `abstract-counterexample-search` returns $\bot$, then the procedure `analyze-partial-safety-proof` is called for learning the reason for the conflict. The learned transformer is given by *learn*, which is same as *AUnit*. However, if `analyze-partial-safety-proof` returns $\top$ and there are no further cases to be explored, hence ACDLS terminates. The invariant of the ACDLS algorithm is that the transformer $f^G_{aunsafe}$ is a sound overapproximation of the concrete unsafe trace transformer f^G_{unsafe}.

Figure 6.22 gives the communication between the Abstract-Counterexample-Search (shown on the left box) and Analyze-Partial-Safety-Proof (shown on the right box) procedures of the ACDLS algorithm. The conflicting elements obtained from the conflict are passed from Abstract-Counterexample-Search to Analyze-Partial-Safety-Proof. On the other hand, a learned transformer *learn* is transferred from Analyze-Partial-Safety-Proof to Abstract-Counterexample-Search which refines the transformer $f^G_{aunsafe}$. ACDLS backjumps after learning using the procedure `backjump`. The backjump is asserting if the learned transformer becomes *unit* after backjumping, that is, the application of *learn* transformer generates new deductions which guides the search away from the conflicting region. In Figure 6.22, the output of ACDLS is *UNSAFE* if $f^G_{aunsafe}$ is not globally bottom. However, the output is *SAFE* if $f^G_{aunsafe}$ is globally bottom.

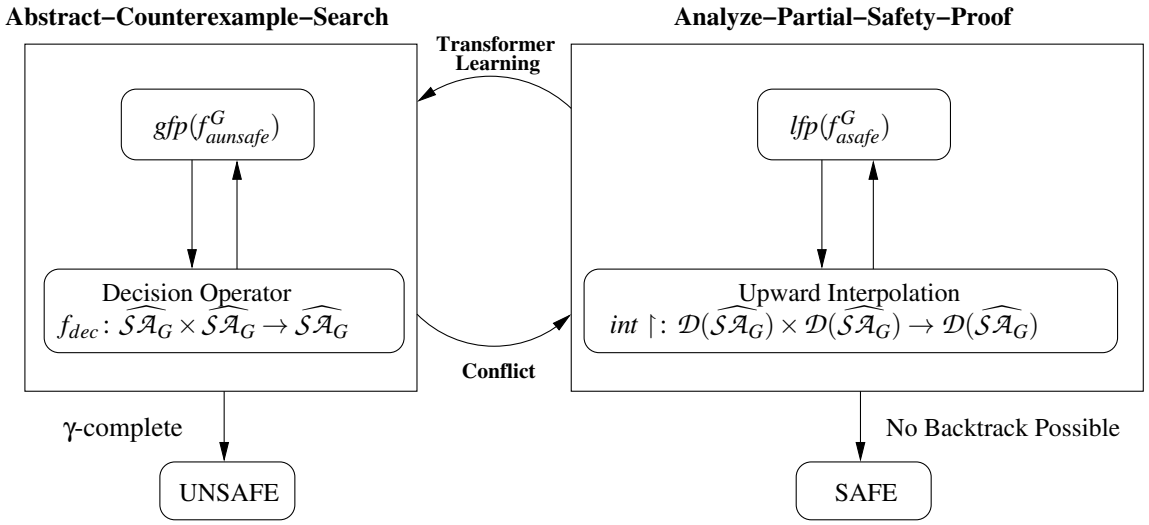


Figure 6.22: Abstract interpretation framework for generalizing CDCL to Safety Verification

Algorithm 6: backjump $(learn, f^G_{aunsafe}, \mathcal{T})$

input : $learn \in \widehat{\mathcal{SA}_G}, f^G_{aunsafe} : \widehat{\mathcal{SA}_G} \to \widehat{\mathcal{SA}_G}, \mathcal{T}$: elements of $\widehat{\mathcal{SA}_G}$
output : $\mathcal{T}$ after backtracking

$f^G_{aunsafe} \leftarrow f^G_{aunsafe} \sqcap learn$
do
  $\mathcal{T} \leftarrow \text{pop}(\mathcal{T})$
while $\gamma_{SA}(f^G_{aunsafe}(\bigsqcap \mathcal{T})) = \bot$
return $\mathcal{T}$

Backjumping

Algorithm 6 gives the `backjump` procedure. The procedure `backjump` removes elements from $\mathcal{T}$ as long as $\gamma_{SA}(f^G_{aunsafe}(\bigsqcap \mathcal{T})) = \bot$ holds true. The function `pop` is used to remove the top element of the trail $\mathcal{T}$. The algorithm returns a modified $\mathcal{T}$ which resets the state of $\mathcal{T}$ to the state where $\gamma_{SA}(f^G_{aunsafe}(\bigsqcap \mathcal{T})) \neq \bot$ holds. If $\mathcal{T}$ is empty after backjumping, then $f^G_{aunsafe}$ is globally bottom.

Asserting Backjump

Let $\mathcal{T}$ and $\mathcal{T}'$ be the trails before and after backjumping, respectively. The function `backjump`$(learn, f^G_{aunsafe}, \mathcal{T})$ is *asserting* if the output trail $\mathcal{T}'$ returned by `backjump` satisfies either of the following two conditions.

1. $\mathcal{T}' = \langle\rangle$, that is, $\mathcal{T}'$ is empty.
2. $learn(\bigsqcap \mathcal{T}')$ and $(\bigsqcap \mathcal{T})$ are not comparable.

Condition (2) above denotes that asserting backjumping drives the abstract counterexample search away from the conflicting region, that is, the trail $\mathcal{T}'$ obtained after learning through *learn* transformer contains newly deduced elements that drive the search into new region of the search space. We now show that $learn(\bigsqcap \mathcal{T}')$ and $(\bigsqcap \mathcal{T})$ are unordered in the lattice $\widehat{\mathcal{SA}_G}$. We denote the transformer after learning as $f^{G\dagger}_{aunsafe}$, where $f^{G\dagger}_{aunsafe} = f^G_{aunsafe} \sqcap learn$.

Lemma 6.9.1. A run of ACDLS algorithm with asserting backjumping maintains the state of the trail in a way that the abstract element $\bigsqcap \mathcal{T}$ obtained from the trail $\mathcal{T}$ before backjumping is incomparable to the abstract element $f^{G\dagger}_{aunsafe}(\bigsqcap \mathcal{T}')$ from trail $\mathcal{T}'$ after backjumping.

Proof. Given $\mathcal{T}$ and $\mathcal{T}'$ represent the trail before and after backjumping, respectively. Let $a = \bigsqcap \mathcal{T}$ and $a' = \bigsqcap \mathcal{T}'$. Recall that `analyze-partial-safety-proof` returns a

tuple (d,t), where $t \sqsupseteq_{SA} a$. Note that the transformer *learn* is same as *AUnit*. Let l be the unit literal in clause t after backjumping. Also, let l and $\bar{l}$ be meet irreducibles of $\widehat{\mathcal{SA}_G}$ where $\bar{l}$ is the complement of l. Hence, the generalized unit rule transformer $AUnit_t$ with respect to the clause t returns $AUnit_t(a') = \bar{l}$ after backjumping. It follows that $f^{G\dagger}_{aunsafe}(a') \sqsubseteq_{SA} \bar{l}$ and $l \sqsupseteq_{SA} t$. Since, $l \sqsupseteq_{SA} t$ and $t \sqsupseteq_{SA} a$, this implies that $l \sqsupseteq_{SA} a$. We decompose the proof into two separate cases, each of which are proven by contradiction.
Case 1: Assume that $(\bigsqcap \mathcal{T}) \sqsubseteq_{SA} f^{G\dagger}_{aunsafe}(\bigsqcap \mathcal{T}')$. That is, $a \sqsubseteq_{SA} f^{G\dagger}_{aunsafe}(a')$. We can infer the following.

1. $a \sqsubseteq_{SA} f^{G\dagger}_{aunsafe}(a')$ implies $a \sqsubseteq_{SA} \bar{l}$, since $f^{G\dagger}_{aunsafe}(a') \sqsubseteq_{SA} \bar{l}$,
2. $a \sqsubseteq_{SA} t$ and $t \sqsubseteq_{SA} l$ implies $a \sqsubseteq_{SA} l$.

Thus, $\gamma_{SA}(a) = \bot$. This is a contradiction since `abstract-counterexample-search` never returns an empty trail, that is, a trail representing an empty set. We now show the other side of the proof.
Case 2: Assume that $f^{G\dagger}_{aunsafe}(\bigsqcap \mathcal{T}') \sqsubseteq_{SA} (\bigsqcap \mathcal{T})$, that is, $f^{G\dagger}_{aunsafe}(a') \sqsubseteq_{SA} a$, we can infer the following.

1. From $a \sqsubseteq_{SA} l$, we can infer $f^{G\dagger}_{aunsafe}(a') \sqsubseteq_{SA} l$.
2. We already know that $f^{G\dagger}_{aunsafe}(a') \sqsubseteq \bar{l}$.

From the above, $\gamma_{SA}(f^{G\dagger}_{aunsafe}(a')) = \bot$. This is a contradiction since `backjump` never returns a conflicting trail.
Thus, we conclude that $(\bigsqcap \mathcal{T}) \not\sqsubseteq_{SA} f^{G\dagger}_{aunsafe}(\bigsqcap \mathcal{T}')$ and $f^{G\dagger}_{aunsafe}(\bigsqcap \mathcal{T}') \not\sqsubseteq_{SA} (\bigsqcap \mathcal{T})$. Hence, $(\bigsqcap \mathcal{T})$ and $f^{G\dagger}_{aunsafe}(\bigsqcap \mathcal{T}')$ are unordered in the lattice $\widehat{\mathcal{SA}_G}$. □

Lemma 6.9.2. For a completely additive and reductive transformer f^{G}_{unsafe} on the powerset of traces $(\mathbb{P}(\Pi), \subseteq)$, let $f^{G}_{aunsafe} \colon \widehat{\mathcal{SA}_G} \to \widehat{\mathcal{SA}_G}$ be a sound overapproximation of f^{G}_{unsafe}. Let $f^{G}_{asafe} \colon \mathbb{D}(\widehat{\mathcal{SA}_G}) \to \mathbb{D}(\widehat{\mathcal{SA}_G})$ be a sound underapproximation of a completely multiplicative and extensive transformer f^{G}_{safe}. Given an abstract element $a \in \widehat{\mathcal{SA}_G}$, if $gfp(f^{G}_{aunsafe})(a) = \emptyset$, then $f^{G}_{aunsafe} \sqcap learn$ is a sound overapproximation of f^{G}_{unsafe}.

Proofsketch. The procedure `analyze-partial-safety-proof` in ACDLS computes a least fixed point of the transformers $\widehat{apost}_{\Sigma^\dagger}$ and $\widehat{apre}_{\Sigma^\dagger}$. The transformer $f^{G}_{aunsafe} \sqcap learn$ preserves the error reachability, that is, learning new transformers from the conflict does not prevent the reachability of the error location. This can be explained as follows. Let $\mathcal{T}$ be the trail which leads to $\bot$ and $\mathcal{T}'$ be the trail after backjumping, then for the given abstract element a, if $gfp(f^{G}_{aunsafe})(a) = \emptyset$, then $\sqcap\mathcal{T}$ is incomparable to $(f^{G}_{aunsafe} \sqcap learn)(\sqcap\mathcal{T}')$

following Lemma 6.9.1. Thus, the transformer $f^{G\dagger}_{aunsafe}$ drives the search away from the conflicting region by preserving the rechability of the error location. Hence, the transformer $f^{G}_{aunsafe}$ is a sound overapproximation of f^{G}_{unsafe}. □

Putting it all together

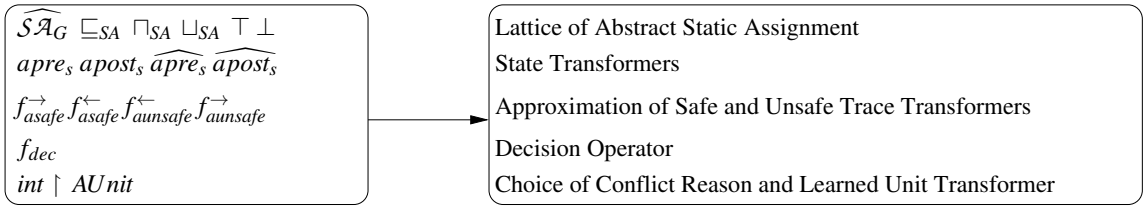


Figure 6.23: Building blocks for generalizing CDCL to safety verification

Figure 6.23 gives the basic building blocks for generalizing the CDCL architecture to safety verification using the abstract interpretation framework. Elements of $\widehat{\mathcal{SA}_G}$ forms the lattice of approximation of program traces ordered by $\sqsubseteq_{SA}$ where each element $a \in \widehat{\mathcal{SA}_G}$ represents the memory state of the program. Each statement s in the program defines four transformers, strongest postcondition $post_s$, existential precondition pre_s, universal postcondition $\widehat{post_s}$, and weakest precondition $\widehat{pre_s}$. Model search computes an overapproximation of unsafe trace transformer using fixed point over $f^{\leftarrow}_{aunsafe}$, $f^{\rightarrow}_{aunsafe}$. The decisions operator is given by f_{dec} which jumps under the greatest fixed point. The conflict analysis computes an underapproximation of safe trace transformer using fixed point over $f^{\leftarrow}_{asafe}$, $f^{\rightarrow}_{asafe}$. The heuristic choice to pick a conflict reason is modelled as upwards interpolation $int \restriction$. Learning overapproximates the unsafe trace transformer while still preserving the error reachability and is parameterized by an element of downset abstract domain $\mathbb{D}(\widehat{\mathcal{SA}_G})$.

6.10 Soundness, Completeness and Termination of ACDLS

We now present the *soundness*, *completeness* and *termination* proof of the ACDLS algorithm. Recall that given a program P with bounded loops and finite recursion depth, a loop-free unrolled program is obtained through complete unrollings of all loops, which is then translated into a set of constraint, $\Sigma^{\dagger}$ and represented by a safety formula, φ. The formula φ is a conjunction of constraints $\sigma \in \Sigma^{\dagger}$ which are derived from P through syntactic translation steps. Furthermore, we consider a finite abstract domain that represents the lattice of abstract static assignments.

Soundness and Completeness of ACDLS

Soundness with respect to safety verification guarantees that if ACDLS returns *safe*, then the program is indeed safe. However, soundness with respect to safety refutation guarantees that if ACDLS returns unsafe, then the program is unsafe. The above two together imply that ACDLS returns *safe* if and only if the program is safe.

Theorem 6.10.1. (Soundness and Completeness). If ACDLS returns $\bot$, then f^G_{unsafe} is *globally bottom*. If ACDLS returns a counterexample, then f^G_{unsafe} is *not globally bottom*.

Proof. Assume ACDLS returns $\bot$. Then, `analyze-partial-safety-proof` procedure returns $\top$. We show that f^G_{unsafe} is *globally bottom*. From Lemma 6.9.2, $f^G_{aunsafe}$ is a sound overapproximation of f^G_{unsafe} at every iteration of the ACDLS algorithm, that is, when `abstract-counterexample-search` returns $(\bot, a)$, then $f^G_{unsafe}(\gamma_{SA}(a)) = \bot$. Given a, a conflict reason r is computed such that $r = \alpha_D \circ \gamma_{SA}(a)$. The relation between a and r can be established by duality given by, $f^G_{safe}(\bot) \supseteq \gamma_{SA}(a) \supseteq \gamma_D(r)$. The procedure `analyze-partial-safety-proof` in Algorithm 4 takes r as input and returns r' where $r, r' \in \mathbb{D}(\widehat{\mathcal{SA}_G})$ such that $r' \sqsupseteq^{\dagger}_{SA} r$ and $\gamma_D(r') = \top$. By fixed point transfer theorem, $\gamma_D(r') = \top$ implies $f^G_{safe}(\gamma_D(r)) = \top$.

From $f^G_{safe}(\gamma_D(r)) = \top$ and $f^G_{safe}(\bot) \supseteq \gamma_{SA}(a) \supseteq \gamma_D(r)$, we can conclude that $f^G_{safe}(\bot) = \top$. Thus, dually it can be shown that $f^G_{unsafe}(\top) = \bot$. Hence, when ACDLS returns $\bot$, f^G_{unsafe} is *globally bottom*.

Assume ACDLS returns a counterexample $\mathcal{C}$. Then, the model search procedure `abstract-counterexample-search` returns (non-$\bot, \mathcal{C}$). This implies that $f^G_{aunsafe}$ is γ-complete at $\mathcal{C} \in \widehat{\mathcal{SA}_G}$, that is, $\gamma_{SA}(\mathcal{C}) \neq \bot$ and $f^G_{aunsafe}(\mathcal{C}) = \mathcal{C}$. Since, $f^G_{aunsafe}$ is γ-complete at $\mathcal{C}$, it implies that $\gamma(f^G_{aunsafe}(\mathcal{C})) = f^G_{unsafe}(\gamma_{SA}(\mathcal{C})) \neq \bot$ (see Definition 6.3.2). Hence, f^G_{unsafe} is not globally bottom. □

Correctness of ACDLS

Partial correctness of ACDLS is trivial since the algorithm either terminates with a model of φ (P is *unsafe*) or deduces $\bot$ (P is safe). It only remains to guarantee termination of ACDLS. The termination is guaranteed by three key conditions – *Progress*, *Finite Closure*, and *Absence of deadlock*. The following theorem shows that given these three conditions, ACDLS always terminates.

Theorem 6.10.2. (Termination). Given a finite abstract domain D, any derivation of ACDLS starting from an initial abstract element $\top$ that contains the set of all traces, will terminate either with $\bot$ or non-$\bot$.

Proof. The number of learned transformers over a finite domain D containing finite number of meet irreducibles is finite. A deadlock freedom property guarantees that only new learned clauses can be generated whenever a conflict takes place. Let us assume that there is a derivation that does not terminate, that is, we must execute learning infinitely often. However, this is a contradiction due to the following properties of the *learn* transformer and the underlying domain D.

1. Learning adds new transformers which are obtained from *asserting backjumps*. An asserting backjump property guarantees that same transformer is never learned twice.

2. The set of learned transformers are contained in the *finite closure* of the set of transformers in $\Sigma^{\dagger}$.

The above conditions guarantee that `analyze-partial-safety-proof` always terminates over a finite lattice.

Similar to a CDCL solver, the progress condition in ACDLS is that the learning causes `abstract-counterexample-search` to deduce new information by navigating the search away from the conflicting element. Following Lemma 6.9.1, if backjumps in ACDLS are asserting, then ACDLS makes progress.

The progress condition is explained by defining a partial order on trails, similar to [83]. The trail $\mathcal{T}$ in ACDLS contains two different kinds of elements – decision (q) and propagations (p). We define a *cost* function that maps elements of the trail into the set $\{1,2\}$, such that $cost(q) = 1$ and $cost(p) = 2$. The intuition here is that the cost of propagations are higher than the cost of decisions. We define $\mathcal{T} \prec \mathcal{T}'$ based on lexicographic ordering on the cost of the trail elements, which is shown below.

$$a.\mathcal{T} \prec b.\mathcal{T}' = cost(a) < cost(b) \vee (cost(a) = cost(b) \wedge \mathcal{T} \prec \mathcal{T}')$$

Clearly, an empty trail is the minimal element, that is, $\langle\rangle \prec \mathcal{T}$ if $\mathcal{T} \neq \langle\rangle$. Adding a new element (a) to the trail $\mathcal{T}$ increases its cost or makes it *bigger* with respect to the partial order $\prec$, that is, $\mathcal{T} \prec \mathcal{T}.a$. This condition holds true for decision and propagation steps. When the procedure backjumps, then the configuration of trail changes from $\mathcal{T}.q.Z$ to $\mathcal{T}.p$, where p is obtained from the application of $f^{G\dagger}_{aunsafe}$ after backjumping. In this case, $cost(q) \prec cost(p)$ and hence the trail is bigger with respect to $\prec$ after backjumping. Thus, each step of ACDLS produces a bigger trail with respect to the partial order and hence ensures progress. However, the trail does not increase forever and thus guarantees the termination of `abstract-counterexample-search` procedure over a finite lattice.

Thus, the progress condition guarantees that `abstract-counterexample-search` terminates, that is, there is *no* infinite sequence of decisions (f_{dec}) and deductions, ($f^{G}_{aunsafe}$).

Starting from $\top$, $f^{G}_{aunsafe}$ deduces sequence of abstract values in the lattice of abstract static assignment $\widehat{\mathcal{SA}_G}$, which follow a downward iteration sequence. The deduction sequence terminates either in $\bot$ (conflict) or some abstract value $a^\dagger \in \widehat{\mathcal{SA}_G}$ such that the set of constraints $\Sigma^\dagger$ is γ-complete in $a^\dagger$. In the former case, ACDLS enters into the conflict analysis phase where it analyses the partial safety proof. However, in the later case, ACDLS terminates with *unsafe* and returns an abstract element $a^\dagger$ where $\gamma_{SA}(a^\dagger) \neq \bot$.

□

6.11 Conclusion

The lattice theoretic account of the CDCL algorithm provides a natural extension to other domains. In this chapter, we present an abstract interpretation framework for generalizing CDCL to precise safety verification over non-distributive template-based abstract domains. The generalization combines overapproximation of greatest fixed point and underapproximation of least fixed point to precisely reason about disjunctive properties over non-distributive domains. To this end, we present a suitable trace-based abstraction for programs with bounded loops using SSA formulas, that satisfy properties for lifting CDCL to reason about correctness of program traces. We then present a CDCL-style algorithm for safety analysis over template-based abstract domain called Abstract Conflict Driven Learning for Safety. We show that learning in ACDLS automatically infers partitions of program traces that are just precise enough to prove the properties that requires disjunction. One of the practical benefits of our work is a novel learning-based technique to automatically improve the precision of fixed point analysis over the non-distributive abstract domains for bounded property verification.

Chapter 7

Safety Verification Using ACDLS - Implementation and Experiments

The success of CDCL algorithm for Boolean satisfiability has inspired its adoption in other domains. In Chapter 6, we present an abstraction interpretation framework for generalizing CDCL to precise safety verification over non-distributive abstract domains called ACDLS. ACDLS alternates between *abstract model search*, which performs overapproximate deduction, and *abstract conflict analysis*, which performs underapproximate abduction with heuristic choice. In this chapter, we instantiate the model search and the conflict analysis algorithms over an abstract domain of the *template polyhedra*, strictly generalizing CDCL from the Boolean lattice to richer non-distributive lattice structures. Our template polyhedra abstract domain can express Intervals, Octagons, Zones, Equality and fixed-coefficient polyhedral constraints. We have implemented ACDLS for automatic bounded safety verification of the software netlist designs (in C) generated from the hardware RTL and other complex softwares. We evaluate the performance of our analyser by comparing with CBMC, which uses Boolean CDCL, and Astrée, a commercial abstract interpretation tool. In this chapter, we refer to a software netlist as a program.

Claim of Novelty In this chapter, we make the following novel contributions.

1. A practical instantiation of ACDLS over the template polyhedra abstract domain, for bounded safety verification. The proposed instantiation is embodied in our tool, *ACDLS*.

2. An abstract model search procedure with the best parameterized abstract transformer that operates over the template polyhedra abstract domain. The parameterization helps the transformer to guide the abstract model search in forward, backward and multi-way directions.

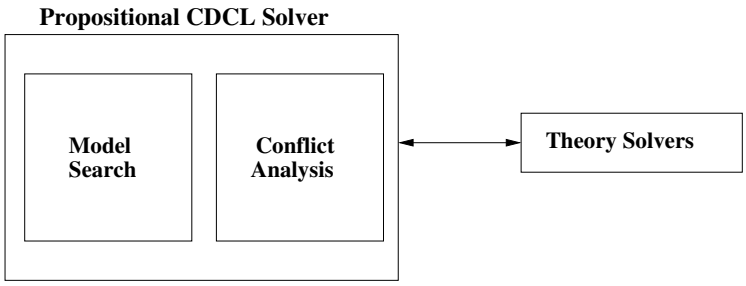


Figure 7.1: Architecture of Classical DPLL(T)

3. An abstract conflict analysis procedure that performs UIP-based transformer learning over the template polyhedra abstract domain.

4. An experimental evaluation to analyze the performance of ACDLS in comparison with a bit-level bounded model checker, CBMC, and a commercial abstract interpretation tool, Astrée.

Plan of the chapter The rest of the chapter is organized as follows. An overview of different solver architectures is presented in Section 7.1. Section 7.2 gives a motivating example. The abstraction domain instances of template polyhedra is described in Section 7.3. The program representation is given in Section 7.4. Section 7.5 gives the details of the ACDLS framework instantiated over template polyhedra abstract domain. The abstract model search and abstract conflict analysis is explained in Section 7.6 and Section 7.7. Section 7.8 presents the experimental results. The benefit of ACDLS over classical abstract interpretation is explained in Section 7.9. Section 7.10 concludes the chapter.

7.1 Overview of Solver Architectures

We discuss various architectures of DPLL based satisfiability solvers. From an architectural standpoint, the following classification of DPLL family of solvers gives an overview of the various similarities and differences between them.

CDCL

The CDCL solver has two key components – *model search* and *conflict analysis*. This is explained in Section 2.2.1.

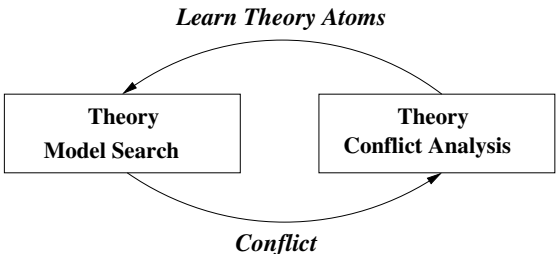


Figure 7.2: Architecture of Natural Domain SMT Solver

DPLL(T)

Figure 7.1 gives the architecture of a DPLL(T) solver [21, 147, 177], also called *Lazy SMT*. DPLL(T) uses a CDCL solver as a black box which is used to enumerate the assignments of the boolean abstraction of the formula. The candidate Boolean assignment is then checked with a solver for the conjunctive fragment of the theory to check satisfiability of quantifier-free formula. Thus, decision-making and learning in DPLL(T) is delegated to the propositional CDCL solver which is used to enumerate theory facts. The theory solver adds blocking clauses in terms of the existing atoms that represent a partial assignment infeasible in the theory. The work of [13] introduced *Splitting on demand* which encodes new theory facts as propositional variables and adds them to a formula. There are various first-order theories such as arithmetic, bit-vector, arrays, uninterpreted functions, and datatypes.

Natural Domain SMT

Figure 7.2 gives the architecture of a natural domain SMT solver, proposed by Cotton et al., that integrates the theory solvers inside the CDCL procedure with techniques such as theory propagation and theory learning. The unification of the theory solver and CDCL search into a common architecture enables direct search of a model of the formula over the natural domain of variables. This led to the development of new solver procedures that perform direct model construction complemented with conflict resolution in some restricted first-order domains such as floating-point [32], linear real arithmetic [65, 160], linear integer arithmetic [134], and nonlinear arithmetic [135].

Generalised DPLL

McMillan et al. [177] presented a Generalised DPLL procedure called *GDPLL* which performs decisions and learning directly in theory. A key difference between the lazy SMT and GDPLL is that the theory deductions in GDPLL only occur in response to a conflict in model search, in contrast to the lazy SMT which restricts theory deductions only in response

to satisfying partial assignments. Additionally, GDPLL allows theory learning in contrast to lazy SMT approach which cannot learn theory facts. Similar theory-specific approaches are presented for equality [8] and integer linear arithmetic [134].

Model Constructing Satisfiability Calculus

While natural domain SMT procedures are quite effective in their respective first-order theories, they have limitations in pure boolean reasoning and are not compatible with the DPLL(T) procedures. A *model-constructing satisfiability calculus* (mcSAT) [83, 133] extends the DPLL(T) procedure through *model assignment* which allows decisions on theory variables. Furthermore, it allows propagations and learning in terms of theory variables that are not present in the given formula.

ACDLS versus DPLL/CDCL-based solvers

There are few fundamental differences between ACDLS and solver procedures discussed above. ACDLS is different from the DPLL(T) procedure in that it does not contain a separate propositional solver and the reasoning happens directly over an abstract domain. ACDLS is also different from GDPLL. This can be explained with an example borrowed from [160]. Consider the formula, $\phi = (a < b) \wedge (a < c) \wedge (b < d \vee c < d) \wedge (d < a)$, where a, b, c, d are integers. GDPLL determines that ϕ is unsatisfiable using decisions and the shadow rule for conflict analysis [160]. By contrast, ACDLS does not need to make any decisions and can prove unsatisfiability of ϕ through deductions in the abstract domain of Inequality. An Inequality abstract domain contains only conjunctions of inequalities. The abstract value computed through evaluating each clause of ϕ over the Inequality abstract domain is given by $\pi = \{a < b, a < c, d < a\}$. The greatest fixed point computation during the abstract model search phase of ACDLS gives the final abstract value $\pi' = \pi \cup \{d < b, d < c\}$. The clause $(b < d \vee c < d)$ of the formula ϕ is unsatisfiable under π'. Hence ϕ is unsatisfiable.

One of the fundamental differences between ACDLS and mcSAT is that the decisions in mcSAT are restricted to concrete model assignments to theory variables and boolean assignments to theory atoms in the formula. On the other hand, decisions in ACDLS can be expressed over set of meet irreducibles representable by the underlying abstract domains. For example, a decision in Interval domain for the branching variable x can be of the form $x \leq 5$. Similarly, a decision in Octagon domain for branching variables x, y can be of the form $x + y \leq 5$.

Another difference between ACDLS and the solver procedures discussed above is based on the encoding of the input formula. ACDLS directly operates on the Static Single Assignment (SSA) representation where each SSA constraint is obtained directly from

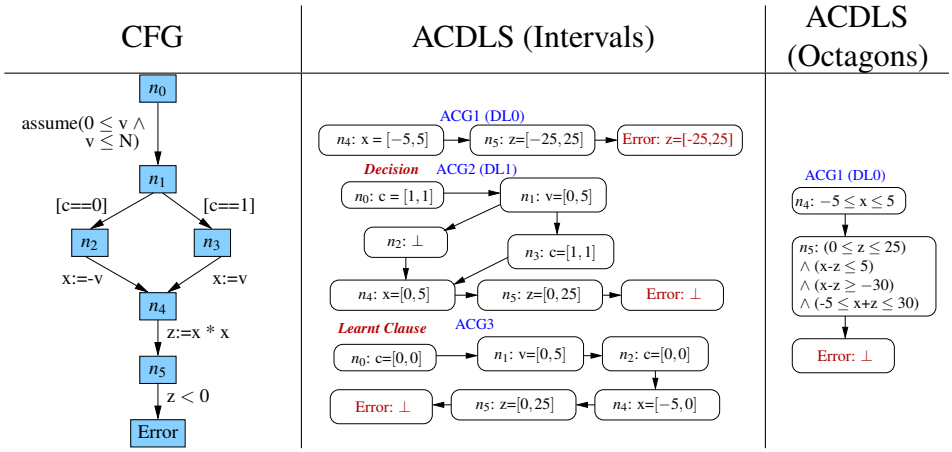


Figure 7.3: CFG and corresponding Abstract Conflict Graphs for Intervals and Octagons

program statements and constitutes an equality or inequality (obtained from assignment statements, conditional expressions, or an assertion) or their disjunctions (obtained from control-flow branches). On the other hand, a DPLL(T) based solver operates on a formula which is a conjunction of sentences in a given logic where a sentence (or clause) is a disjunction of theory atoms. This formula is usually specified in SMT-LIB2 [1] language. For example, a sentence in the logic of quantifier-free linear rational arithmetic (QFLRA) is of the form $c_1x_1 + c_2x_2 + \ldots + c_nx_n \diamond c_0$, where c_i is a rational constant and x_i is a rational variable, and $\diamond$ is either $\geq$ or $>$. A model of QFLRA is an assignment of rational values to variables x_i such that it satisfies every clause of the formula.

7.2 Motivating Examples

We present two simple examples to demonstrate the essence of ACDLS for bounded verification. For each one, we apply three analysis techniques: *abstract interpretation* (AI), SAT-based *bounded model checking* (BMC) and ACDLS.

First Example. The simple control-flow graph in Figure 7.3 squares a machine integer of 32-bits and checks whether the result is positive. To avoid overflow, we assume the input v has an upper bound N. This example shows that a) interval analysis in ACDLS is more precise than a forward AI in the interval domain, and b) ACDLS with intervals can achieve a precision similar to that of AI with octagons without employing more sophisticated mechanisms such as trace partitioning [182].

[1] http://smtlib.cs.uiowa.edu/language.shtml

AI versus ACDLS. Conventional forward interval AI is too imprecise to verify safety of this program owing to the control-flow join at node n_4. For example, the state-of-the-art AI tool Astrée requires external hints, provided by manually annotating the code with partition directives at n_1. This tells Astrée to analyse the program paths separately.

ACDLS can be understood as an algorithm to infer such partitions automatically. For the example in Figure 7.3, interval analysis with ACDLS is sufficient to prove safety. The analysis records the decisions and deductions in a *trail* data-structure. The trail can be seen to represent a graph structure called the *Abstract Conflict Graph* (ACG) that stores dependencies between decisions and deductions, similar to the way an *Implication Graph* [21] works in a SAT solver. Nodes of the ACG in the second column of Figure 7.3 are labelled with the CFG location and the corresponding abstract value. Beginning with the assumption that v=[0,5] at node n_1, the intervals generated by forward analysis in the initial deduction phase at *decision level* 0 (DL0) are x=[-5,5] and z=[-25,25]. These do not prove safety, as shown in ACG1. So ACDLS makes a heuristic decision, at DL1, to refine the analysis. With the decision c=[1,1], interval analysis then concludes x=[0,5] at node n_4, which leads to (Error: $\bot$) in ACG2, indicating that the error location is unreachable and that the program is safe when c=[1,1].

Reaching (Error: $\bot$) is analogous to reaching a conflict in a propositional SAT solver. At this point, a clause-learning SAT solver learns a reason for the conflict and backtracks to a level such that the learned clause is *unit*. By a similar process, ACDLS learns that c=[0,0]. That is, all error traces must satisfy c $\neq 1$. The analysis discards all interval constraints that lead to the conflict and backtracks to DL0. ACDLS then performs interval analysis with the learned clause $c \neq 1$. This also leads to a conflict, as shown in ACG3. The analysis cannot backtrack further and so terminates, proving the program safe. Thus, *decision* and *clause learning* are used to infer the partitions necessary for a precise analysis. Alternatively, the octagon analysis in ACDLS—illustrated in the third column of Figure 7.3—can prove safety with propagations only. No decisions are required. Forward AI with octagons in Astrée is also able to prove safety for this example without manual partitioning. However, in general, the analysis using relational domains become expensive for large programs.

Second Example. In this example, we show that octagon analysis in ACDLS is more precise than forward AI in the octagon domain. The CFG in Figure 7.4 computes the absolute values of two 32-bit variables, x and y, under the assumption $(x = y) \vee (x = -y)$.

AI versus ACDLS. Forward AI in the octagon domain infers the octagonal constraint Error: $p \geq 0 \wedge p+q \geq 0 \wedge q \geq 0 \wedge p+x \geq 0 \wedge p-x \geq 0 \wedge q+y \geq 0 \wedge q-y \geq 0$. Clearly this is too imprecise to prove safety since the octagon constraint is obtained by merging

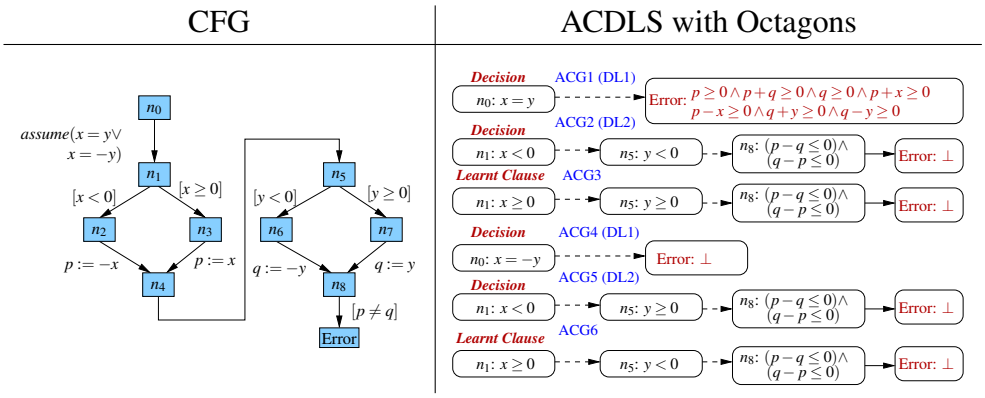


Figure 7.4: CFG and corresponding Abstract Conflict Graphs for Octagon analysis

the control-flow at the join points. The octagonal analysis in ACDLS is illustrated by the ACGs in Figure 7.4. The decision $x = y$ at DL1 is not sufficient to prove safety, as shown in ACG1. So a new decision $x < 0$ is made at DL2, followed by forward propagation that infers $y < 0$ at node n_5. This subsequently leads to safety (Error: $\perp$), as shown in ACG2. The analysis learns the reason for the conflict, discards all deductions in ACG2 and backtracks to DL1. Octagon analysis is run with the learned constraint $(x \geq 0)$ which infers $(y \geq 0)$ at node n_5, as shown in ACG3. This also leads to safety (Error: $\perp$). The analysis now makes a new decision $x = -y$ at DL1. The procedure is repeated leading to results shown in ACG4, ACG5, and ACG6. Clearly, the decisions $x = -y$ and $x < 0$ also lead to safety. The analysis backtracks to DL0 and returns *safe*. Note that the specific decision heuristic we use in this case exploits the control structure of the program to infer partitions that are sufficient to prove safety.

ACDLS versus BMC. ACDLS can require much fewer iterations than SAT-based BMC due to its ability to reason over much richer lattice structures. A SAT-based BMC converts the program into a bit-vector formula and passes it to a CDCL-based SAT solver for proving safety. Table 7.1 compares the statistics for BMC with MiniSAT [2] solver to those for

[2] http://minisat.se/

Solver	Domain	decisions	propagations	conflicts	conflict literals	restarts
Solver statistics for Figure 7.3 (For N = 46000)						
MiniSAT	$BVars \rightarrow \{t, f, ?\}$	233	36436	162	2604	2
ACDLS	$Itvs[NVars]$	1	17	1	1	0
ACDLS	$Octs[NVars]$	0	7	0	0	0
Solver statistics for Figure 7.4						
MiniSAT	$BVars \rightarrow \{t, f, ?\}$	4844	32414	570	4750	5
ACDLS	$Octs[NVars]$	4	412	2	2	0

Table 7.1: SAT-based BMC versus ACDLS for verification of programs in Figure 7.3 and 7.4

Interval	Octagons	Zones	Equality	Fixed-coefficient Polyhedra
$(a \leq x_i \leq b)$	$(\pm x_i \pm x_j \leq d)$	$(x_i - x_j \leq d)$	$(x_i = x_j)$	$(a_1x_1 + \ldots + a_nx_n \leq d)$

Table 7.2: Template instances of template polyhedra abstract domain

interval and octagon analysis in ACDLS. In the column labelled Domains, *BVars* is the set of propositional variables; each of these is mapped to *true* (t), *false* (f) or *unknown* (?). *NVars* is the set of numerical variables; *Itvs*[*NVars*] and *Octs*[*NVars*] are the Interval and Octagon domains over *NVars*. As can be seen, ACDLS outperforms BMC in the total number of *decisions*, *propagations*, *learned clauses* and *restarts* for both example programs.

7.3 Abstract Domains

CDCL SAT solvers operate on the domain of partial assignments [86, 89]. In this chapter, we instantiate ACDLS over a reduced product domain [70] $D[Vars] = \mathcal{B}^{|BVars|} \times \mathcal{TP}[NVars]$ where $\mathcal{B}$ is the Boolean domain that permits abstract values $\{TRUE, \mathsf{false}, \bot, \top\}$ over boolean variables *BVars* in the program, and $\mathcal{TP}$ is a *template polyhedra* [186] domain over the numerical (bitvector) variables *NVars*. Our template polyhedra domain can express various relational and non-relational templates over *NVars*, as given in Table 7.2.

7.3.1 Template Polyhedra Abstract Domain

An abstract value of the template polyhedra domain [186] represents a set X of values of the vector $\vec{x}$ of numerical (bitvector) variables *NVars* of their respective data types. (Currently, signed and unsigned integers are supported.) An abstract value is a constant vector $\vec{d}$ that represents sets of values for $\vec{x}$ for which $C\vec{x} \leq \vec{d}$, for a fixed coefficient matrix C. The domain containing $\vec{d}$ is augmented by a special element $\bot$ to denote the minimal element of the lattice.

Recall the abstraction (α_{SA}) and concretization (γ_{SA}) functions for the abstract static assignment lattice from Definition 6.8.8. In this chapter, we use exactly the same abstraction and concretization functions for the template polyhedra abstract domain. We denote the abstraction function α_{SA} by α and the concretization function γ_{SA} by γ for the template polyhedra abstract domain.

The interval domain is a non-relational domain because a single inequality only contains a single variable. The octagon domain, however, is relational. For a program with $N = |NVars|$ variables, the template matrix C for the interval domain *Itvs*[*NVars*], has $2N$ rows. Hence, it generates at most $2N$ inequalities, one for the upper and lower bounds of each

variable. For octagons $Octs[NVars]$, we have at most $2N^2$ inequalities, one for the upper and lower bounds of each variable and sums and differences for each pair of variables. We now define the various templates supported by our template polyhedra domain.

Interval Abstract Domain The *Interval* abstract domain is a lattice given by $ItvDom = (ItvElm, \sqsubseteq_I, \sqcup_I, \sqcap_I)$, where $ItvElm : (Var \to Itv) \cup \{\bot\}$, and Itv is the set of intervals of type $[l,u]$ over a numeric data type with $l \leq u$. The least element is $\bot$ and the greatest element is $\top$ which maps all variables to their minimum (*min*) and maximum (*max*) values of appropriate types respectively. An interval $\langle x, [min, v]\rangle$ is written as $x \leq v$. The partial order $\sqsubseteq_I$ over elements in the set Itv is given by $I_1 \sqsubseteq_I I_2$ if I_2 contains I_1. A join $(\sqcup_I)$ of two intervals $\langle x_1 \to [l_1,u_1], x_1 \to [l_2,u_2]\rangle$ is an interval $\langle x_1 \to [min(l_1,l_2), max(u_1,u_2)]$. A meet $(\sqcap_I)$ of two intervals $\langle x_1 \to [l_1,u_1], x_1 \to [l_2,u_2]\rangle$ is an interval $\langle x_1 \to [max(l_1,l_2), min(u_1,u_2)]\rangle$. Taking a logical view, the Interval lattice can be seen as closed under conjunction, but not disjunction or negation. For example, the meet is the precise intersection of the values in two intervals, $[2,5] \sqcap_I [4,8] = [4,5]$, while the join is an overapproximation and may contain more elements than the union of the values in the intervals, for example, $[1,5] \sqcup_I [7,10] = [1,10]$.

Octagon Abstract Domain The *Octagon* abstract domain is a lattice which is given by $OctDom = (OctElm, \sqsubseteq_O, \sqcup_O, \sqcap_O)$, where $OctElm : (Var \times Var \to (\mathbb{R} \cup \{\infty\})) \cup \bot$. The least element is $\bot$ that contains all unsatisfiable sets of inequalities and the greatest element is $\top$ which maps the bounds of all octagonal inequalities to ∞. The partial order $\sqsubseteq_O$ over elements in the set $OctElm$ is given by $O_1 \sqsubseteq_O O_2$ iff the bounds of each inequality in O_1 is included (by $\leq$ order) in the bounds of the corresponding inequalities in O_2, that is, the octagons are ordered by the inclusion relations.

The join $(\sqcup_O)$ of two octagons, $\langle (v_i \times v_j \to N_1), (v_i \times v_j \to N_2)\rangle$, is not necessarily an octagon and is computed by taking the piece-wise maximum of bounds of corresponding octagonal inequalities, $\langle (v_i \times v_j \to max(N_1,N_2)\rangle$. However, the meet $(\sqcap_O)$ of two octagons, $\langle (v_i \times v_j \to N_1), (v_i \times v_j \to N_2)\rangle$, is always an octagon and is computed by taking the piece-wise minimum of bounds of corresponding octagonal inequalities, $\langle (v_i \times v_j \to min(N_1,N_2)$. A closed octagon is the smallest octagon following the partial order $\sqsubseteq_O$, among all the octagons that abstract the same concrete values.

The octagon domain is a relational abstract domain that permits $2n^2$ linear inequalities between n program variables. The octagonal inequalities are of two types: binary or unary inequalities as shown below.

$$Binary : \pm v_i \pm v_j \leq a, v_i \neq v_j \qquad Unary : v_i \leq b, \{a,b\} \in \mathbb{R} \cup \infty$$

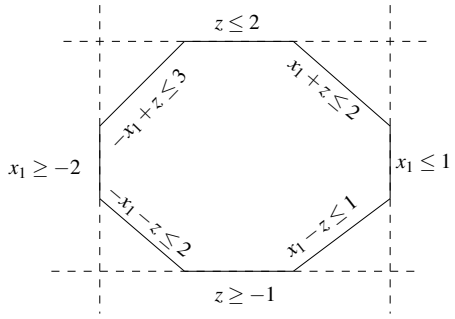


Figure 7.5: Example of an octagon

An element in octagon domain, *OctElm*, is a conjunction of such inequalities and is called an octagon. Figure 7.5 gives an example of an octagon and its associated inequalities. For building program analyzers using octagon domains, the domain also provides a few operators like *widening (*∇*)* and *closure (*$*$*)*. The widening operator is used to accelerate convergence for loops in the program and has a quadratic complexity in the number of variables. If the bound of an octagonal inequality increases every iteration, then ∇ sets the bounds to ∞. However, the closure operator is often used to reduce the degree of overapproximation resulting from the join operation.

Closure Operation in Relation Domains Unlike a non-relational domain, a relational domain such as octagons requires the computation of a *closure* in order to obtain a normal form, necessary for precise domain operations. The closure computes all implied domain constraints. An example of a closure computation for octagonal inequalities is $closure((x - y \leq 4) \wedge (y - z \leq 5)) = ((x - y \leq 4) \wedge (y - z \leq 5) \wedge (x - z \leq 9))$. For octagons, closure is the most critical and expensive operator; it has the cubic complexity in the number of program variables [197]. We therefore compute closures lazily in template polyhedra domain in our abstract model search procedure. The details of lazy closure computation are explained in Section 7.6.4.

Expressiveness of Template Polyhedra A general polyhedral analysis is most expressive but has exponential worst-case space and time complexity [163, 164]. By contrast, template polyhedra are restricted since they can encode inequalities of the form $c_1x_1 + \ldots + c_nx_n \leq d$, where the coefficients $c_1, \ldots, c_n$ are fixed a-priori. Sankaranarayanan et al. [186] showed that the complexity of the template polyhedral analysis is in worst-case polynomial time in the size of the program and the abstract domain used. Figure 7.6 presents a graphical view of the relation between different numerical (relational and non-relational) abstract domains. Thus, the expressivity of template polyhedra lies between weakly relational domains such

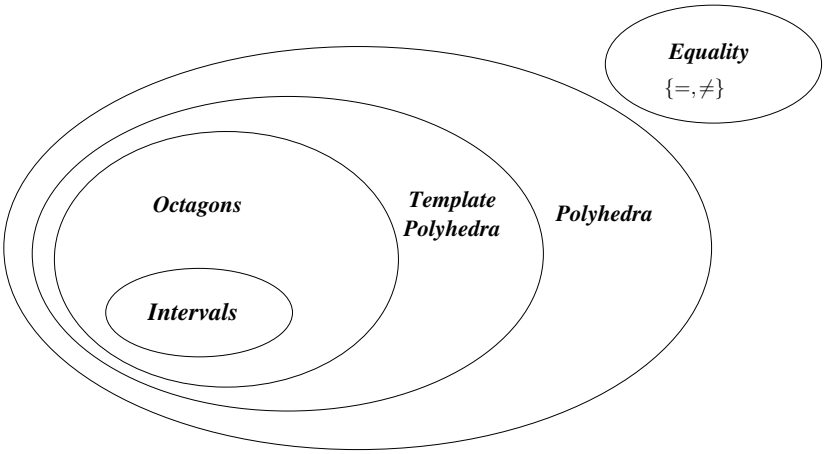


Figure 7.6: Relation between numerical abstract domains

as intervals $(a \leq x_i \leq b)$, octagons $(\pm x_i \pm x_j \leq c)$, and strongly relational domain such as polyhedra domain.

Operations in Template Polyhedra Domain There are several optimisation-based techniques [34, 103, 185] for computing the domain operations, such as meet ($\sqcap$) and join ($\sqcup$), in the template polyhedra domain. In our implementation, we use the strategy iteration approach of [34]. Recall the definition of meet irreducibles, precise complementation property of the template polyhedra domain and meet decomposition from section 6.8.3. We will use these definitions in this chapter for instantiating ACDLS over the template polyhedra abstract domain.

It is worth noting that standard abstract interpretation does not require a complementation operator, so abstract domain libraries, such as APRON [131], do not provide it. But it can be implemented with the help of a meet decomposition operation.

7.3.2 Trace Partitioning

We give a brief description of the trace partitioning abstract domain [182] that is employed by the commerical abstract interpreter Astrée. Trace partitioning is a method for independently analysing the sets of program traces generated by a suitably chosen partitioning function, and was introduced in the context of abstract interpretation as a way to increase the precision of the analysis over the non-distributive abstract domains [27, 123]. However, the use of trace partitioning domain is generally expensive. For example, assuming that the partition is done over the control-flow branches of the program, the total number of environments to be analyzed for each partition of the program control flow is doubled.

The idea of trace partitioning is to distinguish traces using a function $f_{partition} : K \to \Pi$ that maps some set of tokens K to sets of traces Π. We describe two cases based on the value of k. If $|K| = 1$ and $f_{partition}(K) = \Pi$, then there is no discrimination between traces. On the other hand, if $|K| = \Pi$, then each trace is considered in isolation and an analysis with this partition is basically an explicit exploration of all program traces. In practice, static analyzers aim at a partition that falls between the two extremes and systematize the search for a suitable partition using *trace partitioning templates*, which are described next.

Trace Partitioning Templates

The choice of partition function helps to fine-tune the scalability of the analysis by performing coarser splitting where possible and only increasing the number of partitions where necessary. In this section, we briefly mention few partitioning strategies.

Value, control-flow and length-based Partitioning:

Value-based trace partitioning restricts the range of a variable at one or more program locations. For example, if x is a program variable, then the partition of the traces may depend on grouping the input values of the variable x into three separate cases $\{x < 0, x > 0, x = 0\}$. *Control-flow based trace partitioning* distinguishes traces according to control-flow history. Further, *length* or *parity* based trace partitioning is based on the length of a trace. In this case, the partition function may be given by $f_{partition} : \{0,1\} \to \Pi$, where $f_{partition}(0)$ corresponds to those traces that execute the loop an even number of times and $f_{partition}(1)$ corresponds to those traces that execute the loop an odd number of times.

7.4 Program Representation as Constraints

7.4.1 Program Model

In this dissertation, we consider *bounded programs* with safety properties given as a set of assertions, *Assn*. Recall from Section 6.4 that a bounded program is obtained by a transformation that unfolds loops and recursions a finite number of times. The result of this transformation is represented by a set $\Sigma^{\dagger} = Prog \cup \{\neg \bigwedge_{a \in Assn} a\}$, where *Prog* contains an encoding of the statements in the program as constraints, obtained after translating the program into single static assignment (SSA) form via a data flow analysis and $\{\neg \bigwedge_{a \in Assn} a\}$ represents the negation of the conjunction of all assertions $a \in Assn$. We now present the SSA representation of an example program and the corresponding safety formula.

7.4.2 SSA Representation

The SSA representation $\Sigma^{\dagger}$ for the program in Figure 7.3 is given as follows.

$$\begin{array}{l}\{g_0 = (0 \leq v \leq N),\ g_1 = (g_0 \wedge c),\ x_0 = v,\ x_1 = -v,\\ \ x_2 = g_1?x_0 : x_1,\ g_2 = (g_1 \vee g_0 \wedge \neg c),\ z = x_2 \cdot x_2,\ g_2 \wedge z{<}0\}\end{array} \tag{7.1}$$

Assignments such as x:=v become equalities $x_0 = v$, where the left-hand side variable gets a subscripted fresh name. Control flow is encoded using guard variables, e.g. $g_1 = g_0 \wedge c$. Data flow joins become conditional expressions, e.g. $x_2 = g_1?x_0 : x_1$. The assertions in *Assn* are constraints such as $g_2 \Rightarrow z \geq 0$, meaning that if g_2 holds (i.e., the assertion is reachable), then $z \geq 0$ must hold.

Approximation of SSA representations

We briefly discuss various SSA representations of a given program – *Exact*, *Overapproximate* and *Underapproximate*. Recall from Section 6.7.2 that a ϕ node (denoted in SSA encoding as *y#phi*) is used to join data-flow for variables at the loop head. Additionally, our SSA encoding uses special variables such as *guard#ls* and *x#lb* to model a non-deterministic join of data-flow. The variable *guard#ls* is the loop select variable, and the variable *x#lb* encodes loopback.

The various approximations of SSA can be obtained through different assignments to the *phi* node, which is typically of the form $y\#phi = guard\#ls?x\#lb : z;$. Here, the *phi* node *y#phi* is assigned either the loopback variable *x#lb* or some variable z, based on the non-deterministic value of *guard#ls*. If $guard\#ls = 0$, then *y#phi* is assigned a value from above the loop, else it is assigned a non-deterministic value $(x\#lb)$ from below.

We recall the CFG of Figure 6.16 in Figure 7.7 for the purpose of illustrating various SSA representations. The overapproximate and exact SSA encodings of the CFG of Figure 7.7 is given on the left and right side of Figure 7.8 respectively.

Overapproximate SSA semantics

An SSA semantics is *overapproximate* when there is at least one loop in the program that is not fully unwound and every *phi* assigned statements such as $y\#phi = guard\#ls?x\#lb?z;$, are retained in the SSA. Further, the control flow in SSA is modelled implicitly using data variables. This type of representation is typically used for invariant inference [34, 190]. Let us illustrate the overapproximate SSA (shown on the left side of Figure 7.8). Consider the *phi* assigned SSA, $y\#phi12 = guard\#ls14?y\#lb14 : y11;$. Here, the loop select variable $guard\#ls14$ and loopback variable $y\#lb14$ are completely non-deterministic, hence this is an overapproximate representation. When $guard\#ls14 = 0$, then $y\#phi12 = y11$, which

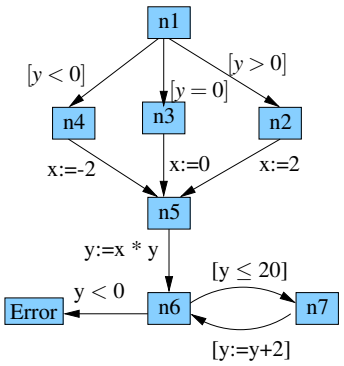


Figure 7.7: An example CFG

Overapproximate SSA	Exact SSA
$\{g0 = TRUE, c4 = y3 >= 0,$ $x5 = -2, g5 = (!c4 \wedge g0),$ $c6 = TRUE, c7 = !(y3 = 0),$ $g7 = (c4 \wedge g0), x8 = 0,$ $g8 = (!c7 \wedge g7), c9 = TRUE,$ $x10 = 2, g10 == (c7 \wedge g7),$ $y11 = x\#phi11 * y3,$ $x\#phi11 = (g10?x10 : (c9 \wedge g8?x8 : x5)),$ $g11 = (c6 \wedge g5 \| c9 \wedge g8 \| g10),$ $y\#phi12 = guard\#ls14?y\#lb14 : y11,$ $c12 = y\#phi12 >= 21,$ $g12 = g11, y13 = 2 + y\#phi12,$ $g13 = (!cond12 \wedge g12),$ $c14 = TRUE, g15 = (c12 \wedge g12),$ $y\#phi12 >= 0 \| !g15\}$	$\{g0 = TRUE, c4 = y3 >= 0,$ $x5 = -2, g5 = (!c4 \wedge g0),$ $c6 = TRUE, c7 = !(y3 = 0),$ $g7 = (c4 \wedge g0), x8 = 0,$ $g8 = (!c7 \wedge g7), c9 = TRUE,$ $c10 = !(y3 >= 1), g10 = (c7 \wedge g7),$ $x11 = 2, g11 = (!c10 \wedge g10),$ $y12 = x\#phi * y3, x\#phi = (g11?x11 :$ $(c10 \wedge g10?x1 : (c9 \wedge g8?x8 : x5))),$ $g12 = (c10 \wedge g10 \| c6 \wedge g5 \| c9 \wedge g8 \| g11),$ $y\#phi = y12, c13 = y\#phi >= 21,$ $g13 = g12, y14 = 2 + y\#phi,$ $g14 = (!c13 \wedge g13), c15 = TRUE,$ $g16 = (c13 \wedge g13), y\#phi >= 0 \| !g16\}$

Figure 7.8: An overapproximate and exact Static Single Assignment representation of CFG of figure 7.7

is the value from above the loop, since $y11$ is assigned between node n5 and n6 in the CFG on the left of Figure 7.8. On the other hand, when $guard\#ls14 = 1$, then $y\#phi12$ gets a non-deterministic value from $y\#lb14$. Clearly, our overapproximate SSA encoding completely breaks all the loops and hence does not encode fixed point semantics. Fixed point semantics would assign $y\#lb14 = y13$, which would have introduced a cyclic behavior. However, note that the overapproximate representation could be strengthened by adding a constraint $y\#lb14 \geq 2 \wedge y\#lb14 \leq 22$.

Exact SSA semantics

An SSA semantics is *exact* if it is obtained through *complete* unwindings of the CFG of a program. The right side of Figure 7.8 gives the exact SSA encoding of the CFG of Figure 7.7. Note that the variables *guard#ls* and *x#lb* are replaced by direct assignments to the *phi* node. Note that replacing the *phi* node is equivalent to setting the variable *guard#ls* to $false$. In this dissertation, we use an exact SSA representation that complete unrolls all loops in a program. Thus, the analysis using ACDLS is sound for programs with bounded loops.

Underapproximate SSA semantics

An SSA semantics is *underapproximate* if it is obtained through *partial* unwindings of loops of a program and every *phi* assigned statement is replaced by $y\#phi = z;$. This type of representation is typically used by the BMC tools. We skip the example of an underapproximate representation since this is straightforward.

7.4.3 Safety Formula

Recall that $\Sigma^\dagger$ denotes the SSA representation of a program. We write *Vars* for the set of variables occurring in $\Sigma^\dagger$. Based on this representation, we define a *safety formula* (φ) as the conjunction of all constraints in $\Sigma^\dagger$, i.e. $\varphi := \bigwedge_{\sigma\in\Sigma^\dagger}\sigma$, similar to Section 6.7.2. Intuitively, φ is the usual SAT encoding of BMC for reachability.

The program is safe if φ is unsatisfiable. The program is unsafe if ACDLS finds a satisfying assignment of φ, which implies that the transformers in $\Sigma^\dagger$ are γ-complete. Note that a satisfying assignment of φ is a counterexample to the assertion $a \in Assn$ where $\Sigma^\dagger = Prog \cup \{\neg \bigwedge_{a\in Assn} a\}$.

Abstract Transformers.

An abstract transformer $[\![\sigma]\!]_D$ transforms an abstract value a through a constraint σ over domain D; it *deduces* information from a and σ. The best transformer is

$$[\![\sigma]\!]_D(a) = a \sqcap \alpha(\{u \mid u \in \gamma(a), u \models \sigma\}) \tag{7.2}$$

where we write $u \models \sigma$ if the concrete value u satisfies the constraint σ. Any abstract transformer that overapproximates the best abstract transformer is a sound transformer and can be used in our algorithm. For example, we can deduce $[\![x = 2(y+z)]\!]_D(a) = (0 \le y \le 2 \wedge 1 \le y-z \le 1 \wedge -2 \le x \le 6)$ for the abstract value $a = (0 \le y \le 2 \wedge 1 \le y-z \le 1)$. We denote the set of abstract transformers for a safety formula φ using the abstract domain D by $\mathcal{A} = \{[\![\sigma]\!]_D \mid \sigma \in \Sigma^\dagger\}$.

Algorithm 7: Abstract Conflict Driven Learning for Safety $ACDLS_{H_P,H_D,H_C}(\mathcal{A})$

input : A program in the form of a set of abstract transformers $\mathcal{A}$.
output : The status safe or unsafe.

$\mathcal{T} \leftarrow \langle\rangle$, $\mathcal{R} \leftarrow []$
$result \leftarrow deduce_{H_P}(\mathcal{A}, \mathcal{T}, \mathcal{R})$
if *result* = conflict **then return** safe
while *true* **do**
  if *result* = sat **then return** unsafe
  $q \leftarrow decide_{H_D}(abs(\mathcal{T}))$
  $\mathcal{T} \leftarrow \mathcal{T} \cdot q$
  $\mathcal{R}[|\mathcal{T}|] \leftarrow \top$
  $result \leftarrow deduce_{H_P}(\mathcal{A}, \mathcal{T}, \mathcal{R})$
  do
    if $\neg analyzeConflict_{H_C}(\mathcal{A}, \mathcal{T}, \mathcal{R})$ **then return** safe
    $result \leftarrow deduce_{H_P}(\mathcal{A}, \mathcal{T}, \mathcal{R})$
  while *result* = conflict
end

7.5 Abstract Conflict Driven Learning for Safety over Template Polyhedra

We describe the *Abstract Conflict Driven Learning for Safety* technique that uses the abstract model search and abstract conflict analysis procedures for bounded property verification of C programs. The model search procedure operates on an overapproximate domain of program traces through repeated application of the abstract deduction transformer, *ded*, and decisions in order to search for a counterexample trace. If the model search finds a satisfying assignment (the corresponding deduction transformer is γ-complete), then ACDLS terminates with a counterexample trace, and the program is *unsafe*. Else, if a conflict is encountered, then it implies that the corresponding program trace is either not valid or safe. ACDLS then moves to the conflict analysis phase where it learns the reason for the conflict from a partial safety proof using an abstract abductive transformer, *abd*, followed by a heuristic choice of conflict reason. ACDLS picks one conflict reason from multiple incomparable reasons for conflict for efficiency reasons. Hence, it operates over an underapproximate domain of conflict reasons. A conflict reason underapproximates a set of invalid or safe traces. The conflict analysis returns a learnt transformer (negation of conflict reason) that overapproximates a set of valid and unsafe traces. Model search is repeated with this new transformer. Else, if no further backtracking is possible, then ACDLS terminates and returns *safe*. We present the ACDLS algorithm in the subsequent section.

The input to ACDLS (Algorithm 7) is a program in the form of a set of abstract

transformers $\mathcal{A} = \{[\![\sigma]\!]_D | \sigma \in \Sigma\}$ w.r.t. an abstract domain D. Recall that the safety formula $\bigwedge_{\sigma \in \Sigma^\dagger} \sigma$ is unsatisfiable if and only if the program is safe. The algorithm is parametrised by heuristics for propagation (H_P), decisions (H_D), and conflict analysis (H_C). The algorithm maintains a propagation trail $\mathcal{T}$ and a reason trail $\mathcal{R}$. The propagation trail stores all meet irreducibles inferred by the abstract model search phase (deductions and decisions). The reason trail maps the elements of the propagation trail to the transformers $ded \in \mathcal{A}$ that were used to derive them.

Definition 7.5.1. The *abstract value* $abs(\mathcal{T})$ corresponding to the propagation trail $\mathcal{T}$ is the conjunction of the meet irreducibles on the trail: $abs(\mathcal{T}) = \bigsqcap_{m \in \mathcal{T}} m$ with $abs(\mathcal{T}) = \top$ if $\mathcal{T}$ is the empty sequence.

The algorithm begins with an empty $\mathcal{T}$, an empty $\mathcal{R}$, and the abstract value $\top$. The procedure *deduce* (details in Section 7.6) computes a greatest fixed point over the transformers in $\mathcal{A}$ that refines the abstract value, similar to the Boolean Constraint Propagation step in SAT solvers. If the result of *deduce* is conflict ($\bot$), the algorithm terminates with safe. Otherwise, the analysis enters into the while loop at line 4 and makes a new decision by a call to *decide* (see Section 7.6.5), which returns a new meet irreducible q. We append q to the trail $\mathcal{T}$. The decision q refines the current abstract value $abs(\mathcal{T})$ represented by the trail, i.e., $abs(\mathcal{T} \cdot q) \sqsubseteq abs(\mathcal{T})$. For example, a decision in the interval domain restricts the range of intervals for variables. We set the corresponding entry in the reason trail $\mathcal{R}$ to $\top$ to mark it as a decision. Here, the index of $\mathcal{R}$ is the size of trail $\mathcal{T}$, denoted by $|\mathcal{T}|$. The procedure *deduce* is called next to infer new meet irreducibles based on the current decision. The model search phase alternates between the decision and deduction until *deduce* returns either sat or conflict.

If *deduce* returns sat, then we have found an abstract value that represents models of the safety formula, $\varphi := \bigwedge_{\sigma \in \Sigma^\dagger} \sigma$, which are counterexamples to the assertion $a \in Assn$ where $\Sigma^\dagger = Prog \cup \{\neg \bigwedge_{a \in Assn} a\}$. Hence, ACDLS returns unsafe. If *deduce* returns conflict, the algorithm enters in the *analyzeConflict* phase (see Section 7.7) to learn the reason for the conflict. There can be multiple incomparable reasons for conflict. ACDLS heuristically chooses one reason C and learns it by adding it as an abstract transformer to $\mathcal{A}$. The analysis backtracks by removing the content of $\mathcal{T}$ up to a point where it does not conflict with C. ACDLS then performs deductions with the learnt transformer. If *analyzeConflict* returns false, then no further backtracking is possible. Thus, the safety formula is unsatisfiable and ACDLS returns safe.

Algorithm 8: Abstract Model Search $deduce_{H_P}(\mathcal{A}, \mathcal{T}, \mathcal{R})$

input : A program in the form of a set of abstract transformers $\mathcal{A}$, a propagation trail $\mathcal{T}$, and a reason trail $\mathcal{R}$.
output : sat or conflict or unknown

$worklist \leftarrow initWorklist_{H_P}(\mathcal{A})$
while $!worklist.empty()$ **do**
  $ded^L \leftarrow worklist.pop()$
  $a \leftarrow ded^L(abs(\mathcal{T}))$
  if $a = \bot$ **then**
    $\mathcal{R}[\bot] \leftarrow ded^L$
    $worklist.clear()$
    return conflict
  else
    $v = onlyNew(a)$
    $\mathcal{T} \leftarrow \mathcal{T} \cdot decomp(v)$
    $\mathcal{R}[|\mathcal{T}|] \leftarrow ded^L$
    $updateWorklist_{H_P}(worklist, v, ded^L, \mathcal{A})$
end
if $\mathcal{A}$ *is* γ*-complete at* $abs(\mathcal{T})$ **then return** sat
return unknown

7.6 Abstract Model Search in Template Polyhedra Domain

Model search in a SAT solver has two steps: *deductions*, which are repeated application of the unit rule (also called Boolean Constraint Propagation, or BCP), to refine current partial assignments, and *decisions* to heuristically guess a value for an unassigned literal. BCP can be seen to compute the greatest fixed point over the partial assignment domain [86]. Below, we present an abstract model search procedure that computes a greatest fixed point over abstract transformers $[\![\sigma]\!]_D$.

7.6.1 Parameterized Abstract Transformers

The key considerations for an abstract transformer are precision and efficiency. A precise transformer is usually less efficient than a more imprecise one. In this chapter, we present a specialised variant of the abstract transformer to compute deductions called *Abstract Deduction Transformer* (ADT), which is parametrised by a given *subdomain* $L \subseteq D$. A subdomain contains a chosen subset of the elements in D including $\bot$ and $\top$ that forms a lattice. The use of a subdomain serves two purposes: a) It allows us to elegantly and flexibly guide the deductions in *forward*, *backward* or *multi-way* direction, which in turn affects the analysis precision, and b) it makes deductions more efficient, for example by performing lazy closure in template polyhedra domain.

An ADT is formally defined as follows.

$$[\![\sigma]\!]^L_D(a) = a \sqcap_D \alpha_L(\{u \mid u \in \gamma_D(a), u \models \sigma\}) \tag{7.3}$$

For $L = D$, the ADT is identical to the abstract transformer defined in Eq. (7.2) in Section 7.4.3. Note that a restricted subdomain makes a transformer less precise but more efficient. Conversely, an unrestricted subdomain make a transformer more precise, but less efficient. Therefore, we have the property $[\![\sigma]\!]^D_D(a) \sqsubseteq [\![\sigma]\!]^L_D(a)$. To illustrate the point (a) above, we give examples that demonstrate how the choice of subdomain influences the propagation direction.

Forward Transformer. For an abstract value $a = (0 \leq y \leq 1 \wedge 5 \leq z)$, a constraint $\sigma = (x = y + z)$, and $L = Itvs[\{x\}]$, we have $[\![x = y+z]\!]^{Itvs[\{x\}]}_{Itvs[\{x,y,z\}]}(a) = a \sqcap (x \geq 6)$. Assuming that the equality $x = y + z$ originated from an assignment to x, this performs a right-hand side (rhs) to left-hand side (lhs) propagation and hence emulates a forward analysis.

Backward Transformer. For an abstract value $a = (0 \leq x \leq 10 \wedge 0 \leq y \leq 1 \wedge 5 \leq z)$, a constraint $\sigma = (x = y+z)$, and $L = Itvs[\{y,z\}]$, we have $[\![x = y+z]\!]^{Itvs[\{y,z\}]}_{Itvs[\{x,y,z\}]} = a \sqcap (z \leq 10)$. This performs an lhs-to-rhs propagation and hence emulates a backward analysis.

Multi-way Transformer. For an abstract value $a = (c \leq 1 \wedge c \geq 1 \wedge x \leq 5 \wedge x \geq 5)$, a constraint $\sigma = ((c = (x = y)) \wedge y = y + 1)$ and $L = Itvs[\{c,x,y\}]$, we have $[\![\sigma]\!]^{Itvs[\{c,x,y\}]}_{Itvs[\{c,x,y\}]} = a \sqcap (y \leq 6 \wedge y \geq 6)$. This performs an lhs-to-rhs propagation for $c = (x = y)$ and rhs to lhs propagation for $y = y + 1$ and hence emulates a multi-way analysis.

7.6.2 Implementation of Abstract Deduction Transformer

Recall the ADT in Equation 7.3, $[\![\sigma]\!]^L_D(a) = a \sqcap_D \alpha_L(\{u \mid u \in \gamma_D(a), u \models \sigma\})$. For simplicity, we assume $L = D$. We will now describe the implementation of the functions, α_L and γ_D in the above equation using symbolic formulas for obtaining the most precise results. We will denote the abstraction and concretization function that operates on symbolic formulas by α_s and γ_s respectively.

The work of [114] presents a technique to construct the best abstract transformer for fixed, finite, cartesian products of Boolean values, also known as *predicate abstraction domain* [10, 11], using decision procedure. Reps et al. [181] showed that the best abstract transformer can also be developed for any arbitrary finite-height abstract domain and is not limited to predicate abstraction domains. The best abstract postcondition transformer for the constraint $\sigma \in \Sigma^\dagger$, denoted by $[\![\sigma]\!]_D$, can be expressed in terms of the concrete postcondition

Algorithm 9: Abstraction function $\alpha_s(\chi')$ for computing best abstract transformer

input : A logical formula χ' representing set of concrete values symbolically
output : Abstract value a in abstract domain D

1 $a \leftarrow \bot$
2 **while** $\exists u \models \chi'$ **do**
  `/* Here α is the abstraction function of` $\mathcal{TP}$ `domain */`
3 $a \leftarrow a \sqcup \alpha(u)$
4 $\chi' \leftarrow \chi' \wedge \neg\gamma_s(a)$
5 **end**
6 **return** a

transformer by, $[\![\sigma]\!]_D(a) = \alpha_s(\{u \mid u \in \gamma_s(a), u \models \sigma\})$. The precision obtainable in a given abstraction is limited by this best abstract postcondition transformer. In this dissertation, we use similar idea presented by Reps et al. [181] for computing the best abstract transformer, but in a more efficient manner. We now define the construction of the α_s and γ_s functions.

The γ_s function

The concretization function, $\gamma_s(a)$, maps an abstract value a to a logical formula χ, such that a and χ represent the same set of concrete values. This allows symbolic manipulation of the concrete values through operations on the formula χ. The advantage of symbolic representation of the result of γ_s function is that it avoids enumerating possibly infinite set of concrete values when a concretization function is applied. It then applies σ to χ at the symbolic level and maps the result back to the abstract domain using the abstraction function α_s, which is defined next. That is, the ADT in Equation 7.2 is recast using χ and σ which operates at the symbolic level such that $[\![\sigma]\!]_D = \alpha_s \circ \sigma \circ \chi$.

Recall that an abstract value in $\mathcal{TP}$ is a constant vector $\vec{d}$ that represents sets of values for $\vec{x}$ for which $C\vec{x} \leq \vec{d}$, for a fixed coefficient matrix C. Thus, the concretization function for the template polyhedra domain $\mathcal{TP}$ simply transforms the vector of values given by $\vec{d}$ into a logical formula.

The α_s function

The abstraction function, α_s, is implemented in a way that represents the most precise abstract value that overapproximates the result of $\sigma \circ \chi$, where χ symbolically represents a set of concrete values. Note that the result of $\sigma \circ \chi$ is expressed symbolically as a logical formula, χ'. Hence, the abstraction function $\alpha_s(\chi')$ is also computed symbolically. A naive way to compute $\alpha_s(\chi')$ proposed by Reps et al. [181] is shown in Algorithm 9.

Discussion of Algorithm 9

The input to the algorithm is a logical formula χ' that is a symbolic representation of a set of concrete values. The output is an abstract value a that represent the most precise overapproximation of the formula χ'. The abstract value a is initialized to $\bot$. The operator $\sqcup$ is the join operation in the abstract domain. Line 2 of Algorithm 9 makes a call to a SAT solver to get the satisfying assignment of χ'. The abstraction function α in line 3 maps concrete value u where $u \models \chi'$ to an abstract value given by $\alpha(u)$. The concretization function γ_s in line 4 is the same as defined above that maps an abstract value to a logical formula.

A major pitfall in the working of Algorithm 9 is that it enumerates all the satisfiable solution of χ' which may be very expensive to compute. However, due to the convex properties of the template polyhedra abstract domain, we can compute $\alpha_s(\chi')$ much more efficiently using binary search approach [34], which is implemented in the tool 2LS [190]. Suppose we want to compute the abstraction of the symbolic formula $f(x)$ for Interval abstract domain, that is, we want to find the tightest value of N such that $\forall x.f(x) \leq N$ is unsatisfiable. To do so, we check whether $\exists x.!(f(x) \leq N)$ is satisfiable using a decision procedure. For $x = \texttt{MAXINT}$, where `MAXINT` is the largest value of integer, the above equation is unsatisfiable, since the bound N is less than `MAXINT`. The binary search solver gives a new value of N, say N_1, such that $\exists x.!(f(x) \leq N_1)$ is satisfiable. We now add the constraint $f(x) > N_1$ to our original formula for the next iteration. Suppose, the binary search solver now returns a new value N_2 such that $\exists x.!(f(x) \leq N_2)$ is unsatisfiable. The binary search solver now returns a value between N_1 and N_2, say $N3$. This process continues until there are consecutive values N_k, N_{k+1}, for which the above query returns satisfiable and unsatisfiable respectively.

An example of best abstract transformer

Consider the abstract value in Interval domain, $Val = \{a = [0,0], b = [1,2], C = \top\}$ and constraint $\sigma = (c = a + b)$, then $\gamma_s(Val)$ produces the logical formula which is given by, $\chi = (a \leq 0 \wedge a \geq 0 \wedge b \geq 1 \wedge b \leq 2)$. Now, the concrete postcondition transformer of $\sigma \circ \chi$ gives the logical formula $\chi' = (a \leq 0 \wedge a \geq 0 \wedge b \geq 1 \wedge b \leq 2 \wedge c = a + b)$. Using Algorithm 9, $\alpha_s(\chi')$ gives the abstract value $Val' = \{a = [0,0], b = [1,2], C = [1,2]\}$ in the Interval domain.

Thus, the concrete domain enables symbolic representation of concrete values and represents the result of the concrete postcondition transformer symbolically. The abstraction

function restricts the result of concrete post transformer to those expressible in the abstract domain.

7.6.3 Algorithm for the Deduction Phase

Algorithm 8 presents the deduction phase *deduce* in the abstract model search procedure. The input to *deduce* is the set of abstract transformers, a propagation trail ($\mathcal{T}$) and a reason trail ($\mathcal{R}$). Additionally, the procedure *deduce* is parametrised by a propagation heuristic (H_P). We write the ADT $[\![\sigma]\!]^L_D$ as ded^L in Algorithm 8. The algorithm maintains a *worklist*, which is a queue that contains ADTs. The propagation heuristic provides two functions *initWorklist* and *updateWorklist*. The order of the elements in the worklist and the subdomain L associated with each ADT (ded^L) determine the propagation strategy (forward, backward, multi-way). These two functions construct a subdomain (L) for ded^L by calling the function *MakeL* such that $L = MakeL_D(V)$, where V are the variables that appear in ded^L. The abstract value a is updated upon the application of ded^L in line 4 in Algorithm 8. The function $onlyNew(a) = \bigsqcap(decomp(a) \setminus decomp(abs(\mathcal{T})))$ is used to filter out all meet irreducibles that are already on the trail in order to obtain only new deductions (v) when applying the ADT (shown in line 10). Depending on the propagation heuristics, *updateWorklist* adds ADTs ded^L to the worklist that contains variables that appear in v, and updates the subdomains of the ADTs in the worklist to include the variables in v (shown in line 13).

If ded^L deduces $\bot$, then the procedure *deduce* returns conflict (shown in line 8). Otherwise, when a fixed point is reached, i.e., the worklist is empty, we check whether the abstract transformers $\mathcal{A}$ are γ-complete [86] for the current abstract value $abs(\mathcal{T})$ (shown in line 15). Intuitively, this checks whether all concrete values in $\gamma(abs(\mathcal{T}))$ satisfy the safety formula φ, where $\varphi := \bigwedge_{\sigma \in \Sigma^\dagger} \sigma$ is obtained from the program transformation (as defined in Section 7.4.1). If it is indeed γ-complete, then *deduce* returns sat. Otherwise, the algorithm returns unknown and ACDLS makes a new decision.

7.6.4 Computing Lazy Closure for Template Polyhedra

Computing the closure for relational domains, such as octagons, is expensive. An advantage of our formalism in Eq. (7.3) is that the *closure* operation for relational domains can be computed in a lazy manner through the construction of a subdomain L. The construction of L allows us to perform one step of the closure operation when ded^L is applied. For example, let us consider $D = Octs[\{x,y,z\}]$ and $V = \{y\}$. An octagonal inequality relates at most two variables. Thus it is sufficient to consider the subdomain $MakeL_D(\{y\}) =$

$Octs[\{y\}] \cup Octs[\{x,y\}] \cup Octs[\{y,z\}]$, which will compute the one-step transitive relations of y with each of the other variables. Only if any subsequent abstract deduction transformer makes new deductions on x or z, then the next step of the closure will be computed through the subdomain $Octs[\{x,z\}]$. Hence, an application of each abstract deduction transformer does not compute the full closure in the full domain, but computes only a single step of the closure in a subdomain. This makes each deduction step more efficient but may require more steps to reach the fixed point.

Let us demonstrate the idea of lazy closure with a concrete example. Assume the program on the left of Figure 7.9, where L1 and L2 denote the program locations. We analyze the program with an octagon domain (*Octs*). The lazy closure computation in the Octagon domain is shown on the right of Figure 7.9.

Recall that in order to compute the normal form of a closure in the Octagon domain, we need to compute all implied constraints among numerical variables. The closure operation is necessary to perform precise domain operations. The analysis in Figure 7.9 performs forward propagation in *Octs* by creating a subdomain L for every transformer using the function *MakeL*. Note that the subdomain consisting of the *lhs* variables of the transformers guides the analysis in forward direction in Figure 7.9. The subdomain corresponding to L1 over y is given by $Octs[\{y\}] \cup Octs[\{y,z\}]$. This means, only those deductions, which are implied by the domain $Octs[\{y\}] \cup Octs[\{y,z\}]$ can be inferred at L1. No deductions over $Octs[\{y,x\}]$ are inferred at L1. Thus, we delay the deductions over the subdomain containing the variables $\{y,x\}$ until we encounter an abstract transformer over these variables. Though the lazy deduction approach does not admit a normal form for octagonal constraints after the application of the transformer at L1, but it makes the deduction step at L1 more efficient.

Assume that the initial abstract value before executing the statement at location L1 is $a = \{x = y\}$. Then, the deduction at L1 infers $\{y = z\}$. For simplicity, we denote the octagonal deductions $\{y - z \leq 0, z - y \leq 0\}$ by the equality constraint $\{y = z\}$. Thus, the updated abstract value is $a = \{x = y \wedge y = z\}$. We now analyze the transformer at L2. The subdomain for L2 over variable x (for forward propagation) is given by $Octs[\{x\}] \cup Octs[\{x,y\}] \cup Octs[\{x,z\}] \cup Octs[\{x,w\}]$. Note that we delayed the deduction over $Octs[\{x,y\}]$ at L1, but only perform the deductions over $Octs[\{x,y\}]$ at L2. The new deductions at L2 are $\{x = z, x - w \leq 1, x - w \geq 1\}$ and the final abstract value is $a = \{x = y \wedge y = z \wedge x = z \wedge x - w \leq 1 \wedge x - w \geq 1\}$. Thus, the normal form over $Octs\{x,y,z\}$ is only achieved at L2. However, we do not perform deductions over $Octs\{w,z\}$ at L2, which is delayed until the point where we encounter an abstract transformer that forces us to infer such deductions. Finally, the full closure in the full domain is computed through the greatest fixed

point iteration in the model search phase. The above example illustrates the lazy closure computation technique in ACDLS for the Octagon domain.

C program	Lazy Closure Computation
int main() { L1: y=z; L2: x=w+1; }	L1: $MakeL_D(\{y\}) = Octs[\{y\}] \cup Octs[\{y,z\}]$ New Deductions: $\{y = z\}$ Abstract Value: $a = \{x = y \wedge y = z\}$ L2: $MakeL_D(\{x\}) = Octs[\{x\}] \cup Octs[\{x,y\}] \cup Octs[\{x,z\}] \cup Octs[\{x,w\}]$ New Deductions: $\{x = z, x - w \leq 1, x - w \geq 1\}$ Abstract Value: $a = \{x = y \wedge y = z \wedge x = z \wedge x - w \leq 1 \wedge x - w \geq 1\}$

Figure 7.9: Lazy closure operation for Octagon domain

7.6.5 Decisions

A decision q is a meet irreducible that refines the current abstract value $abs(\mathcal{T})$, when the result of the fixed point computation through deduction is neither a *conflict* nor a *satisfiable model* of φ. A decision must always be consistent with respect to the trail $\mathcal{T}$, i.e., $abs(\mathcal{T} \cdot q) \neq \bot$. A new decision increases the decision level by one. Given the current abstract value $abs(\mathcal{T})$, the procedure *decide* in Algorithm 7 heuristically returns a meet irreducible expressible by the underlying abstract domain D.

For example, a decision in the Interval domain can be of the form xRd where $R \in \{\leq, \geq\}$, and d is the bound. A decision in the Octagon domain can specify relations between variables, and can be of the form $ax - by \leq d$, where x and y are variables, $a, b \in \{-1, 0, 1\}$ are coefficients, and d is a constant. Section 7.8.3 discusses various decision heuristics in ACDLS.

7.7 Abstract Conflict Analysis in Template Polyhedra Domain

Recall from Section 6.9.2 that the abductive inference in programs find the reason for the conflict by computing an underapproximation of the least fixed point of the weakest precondition transformer. In this section, we present the abstract conflict analysis procedure in ACDLS over the template polyhedra abstract domain.

7.7.1 Abstract Conflict Analysis

A conflict analysis procedure involves two steps: *abduction* [161] and *heuristic choice for generalization*. Abduction infers possible reasons for a conflict, which is followed by heuristically selecting a conflict reason. Below, we define an abstract abductive transformer that gives a set of models that do not satisfy a formula.

Definition 7.7.1. An *abstract abductive transformer*, $abd^D_\varphi : A \to A$, for a formula φ for an input abstract value A in abstract domain D such that $ded^D(A) \sqsubseteq \bot$, is given by, $abd^D_\varphi(A) = \{a \mid a \in A \vee a \not\models \varphi\}$.

An abstract abductive transformer is a dual of abstract deduction transformer, that is, $ded^D(A) = \neg \circ abd^D_\varphi(\neg A)$. Informally, an abstract abductive transformer, abd, for a given formula φ adds abstract models to the input set A that do not satisfy φ.

Example 7.7.1. For example, an abstract abductive transformer for $\varphi \hat{=} \{x = y+1\}$ and $A \hat{=} x \geq 0$ is given by, $abd^{Itvs}_{x=y+1}(x \geq 0) = (x \geq 0 \cup y \leq -2)$.

The trail $\mathcal{T}$ in ACDLS can be seen to represent a graph structure called the *Abstract Conflict Graph* (ACG) that stores dependencies between decisions and deductions. The ACG is defined as follows.

Definition 7.7.2. An Abstract Conflict Graph (ACG) is a directed acyclic graph in which the vertices are defined by deduced elements (including a special conflict node ($\bot$)) or a decision node in the trail $\mathcal{T}$. The edges in ACG are obtained from the reason trail $\mathcal{R}$ that maps pairs of elements $\{a_1, a_2\}$ in $\mathcal{T}$ to the abstract transformer $\sigma \in \mathcal{R}$ such that $[\![\sigma]\!]_D(a_1) = a_2$, that is, σ when applied to a_1 gives the deduced element a_2.

Abstract conflict analysis in ACDLS computes an underapproximation of *abd* by finding an Unique Implication Point (UIP) in ACG. Conflict analysis in propositional solvers performs abduction through clauses stored in the reason trail, and computes a generalization of the original conflict reason. This is achieved by computing a generalized conflict reason for individual deduction steps. For example, consider the partial assignment $\{b : f, a : t, d : f\}$ and the clause $(a \vee b) \wedge (b \vee c)$, which gives the deduction $\{c : t\}$. The conflict analysis procedure may generalize the partial assignment, $\{b : f, a : t, d : f\}$ to $\{b : f, a : t\}$, which still gives the deduction $\{c : t\}$.

Recall from Section 6.9.2 that conflict analysis in ACDLS operates over a downset abstract domain, represented by $\mathbb{D}(D)$. In ACDLS, we assume that each overapproximate deduction transformer $[\![\sigma]\!]_D$ and abstract element $m \in D$ is associated with a local underapproximate transformer, $abdgen^D_{\sigma,m}$, in a downset domain $\mathbb{D}(D)$. We call the transformer $abdgen^D_{\sigma,m}$ as *abductive generalization transformer*. The transformer, $abdgen^D_{\sigma,m} : \mathbb{D}(D) \to \mathbb{D}(D)$, maps an existing reason $a \in \mathbb{D}(D)$ for the deduction m to a more generalized reason $a' \in \mathbb{D}(D)$. That is, the abductive generalization transformer computes the generalized reason for individual deductions.

We present a few examples of abductive generalization transformer over different domain instances of the template polyhedra abstract domain.

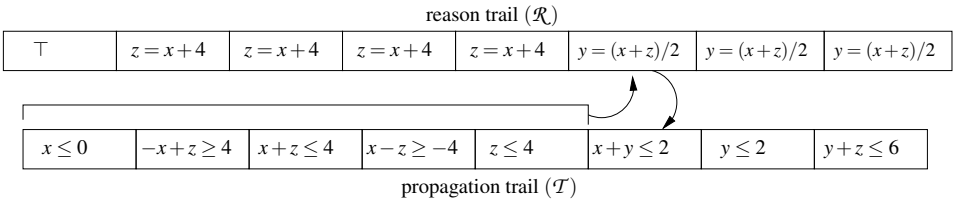


Figure 7.10: Property of Trail in ACDLS

Example 7.7.2. An abductive generalization transformer for the formula $\varphi \hat{=} \{x = y+z\}$ over an Interval domain is given by, $abdgen^{Itvs}_{x=y+z,x\geq 5}(0 \leq y \leq 1 \wedge 5 \leq z) = (y \geq 0 \wedge z \geq 5)$.

Example 7.7.3. An abductive generalization transformer for the formula $\varphi \hat{=} \{x = 2(y+z)\}$ over an octagon domain given by,

$$abdgen^{Octs}_{x=2(y+z),(0\leq y\leq 2\wedge 1\leq y-z\leq 1\wedge -2\leq x\leq 6)}(0 \leq y \leq 2 \wedge 1 \leq y-z \leq 1 \wedge t \geq 1 \wedge t \leq 2) \quad (7.4)$$

is $(0 \leq y \leq 2 \wedge 1 \leq y-z \leq 1)$.

The main idea of abductive reasoning is to iteratively replace an abstract element s in the conflict reason by an abstract value that is sufficient to infer s. Conflict abduction is performed by obtaining cuts through markings in the trail $\mathcal{T}$ using the Unique Implication Point (UIP) search Algorithm [196, 226]. Every cut is a reason for the conflict. The abstract UIP search in ACDLS can be understood as abstract conflict graph cutting algorithm where the vertices of the graph are elements of the domain D.

Property of the Trail in ACDLS

A property of the trail $\mathcal{T}$ in ACDLS is that the meet of the elements in $\mathcal{T}$ up to an arbitrary index i where $a = \{\bigsqcap_{0\leq j\leq i}(a_j) | a_j \in abs(\mathcal{T})\}$, must imply the element $a' \in \mathcal{T}[(i+1)]$ in the subsequent index in $\mathcal{T}$, such that $([\![\sigma]\!]_D)(a) = a'$ holds and $\mathcal{R}[|i+1|] = ([\![\sigma]\!]_D)$.

Note that the trail $\mathcal{T}$ is implemented simply by a C++ vector Standard Template Library (STL), which stores all the deduced meet irreducibles at a particular decision level dl. The reason trail $\mathcal{R}$ is also implemented by a vector where each element of the vector at index i stores the abstract transformer $[\![\sigma]\!]_D$ used to deduce the corresponding element at index i of $\mathcal{T}$, and a pair $\langle i, e\rangle$ containing the indices of the vector $\mathcal{T}$ where i is the index of the first meet irreducible deduced by the transformer $[\![\sigma]\!]_D$ and e is the index of the last meet irreducible deduced by the same transformer at decision level dl, assuming that an application of abstract transformer $[\![\sigma]\!]_D$ deduces $(e-i)+1$ meet irreducibles at the decision level dl.

Example 7.7.4. Figure 7.10 illustrates the property of the trail graphically for the formula $\varphi \hat{=} \{x+y=z \wedge x+z=2y \wedge z+y>10\}$. For the partial abstract value, $a=\{x\leq 0 \wedge -x+z\geq 4 \wedge x+z\leq 4 \wedge x-z\geq -4 \wedge z\leq 4\}$, obtained from the trail, the result of abstract deduction transformer for the constraint, $x+z=2y$, over the Octagon domain is given by $[\![y=(x+z)/2]\!]_{Octs}(a)=\{x+y\leq 2, y\leq 2, y+z\leq 6\}$, which is shown by the arrows in Figure 7.10.

Conflict analysis in ACDLS is different from propositional conflict analysis for two reasons, as listed below.

1. The content of the trail $\mathcal{T}$ obtained from an octagon analysis contains octagonal constraints. Due to the relational nature of these constraints, the same variable may occur in different octagonal constraints which are stored in different indices of the trail $\mathcal{T}$. However, the trail obtained from an interval analysis may contain at most two occurrences of the same variable (upper and lower bound) at any decision level, similar to the Interval Constraint Propagation (ICP) technique proposed by Franzle et al. [97]. By contrast, a trail in the propositional solver contains any variable only once. Furthermore, the unit clause rule in the propositional solver ensures that an antecedent only produce one implied assignment at decision level dl. However, in ACDLS, a transformer in $\mathcal{R}$ may deduce one or more elements at decision level dl.

2. The conflict analysis in propositional solver uses *resolution* [183] operation which infers new propositional clauses. However, ACDCL uses a binary marking algorithm (see Algorithm 12) which identifies the trail elements that are necessary for deriving the conflict and generalizes the marked trail elements to generate a new constraint that is a logical consequence of the original constraints. The conflict analysis in ACDCL is similar to the cutting plane technique [51,110] used by the Pseudo-Boolean solver [194] for inferring new pseudo-boolean constraints.

Algorithm for Abstract Conflict Analysis

Algorithm 10 presents the abstract conflict analysis procedure, $analyzeConflict_{H_C}(\mathcal{A},\mathcal{T},\mathcal{R})$, with abstract first UIP search. The input to the algorithm is the set of abstract transformers, $\mathcal{A}$, the trail $\mathcal{T}$ and reason trail $\mathcal{R}$. The output is *false* if no further backtracking is possible, otherwise the algorithm returns *true*.

The call to the procedure *getConflictClause* in line 2 of Algorithm 10 returns the conflict clause contained in C along with the backjumping level, $blevel$. We briefly describe the procedure *getConflictClause* given in Algorithm 11. The initial conflict reason ϕ_g

contains all the decided meet irreducibles in *dec* up to the decision level *dlevel* − *1* and the meet irreducibles f from the trail segment in *dlevel* that contradict with the transformer at $\mathcal{R}[\bot]$. The meet irreducibles f is obtained using the procedures *getSymbols* and *getMatchingMeetIrreducible*. The initial generalized reason v for the conflict $(\bot)$ corresponding to the transformer at $\mathcal{R}[\bot]$ is obtained using the transformer $abdgen^{D}_{\sigma,\bot}(f)$. Thus, the initial conflict reason ϕ_g is the conjunction of elements in v and *dec*. For each element $\rho \in decomp(\phi_g)$, the procedure *findContradiction* returns the earliest decision level at which ρ contradicts an element of the trail $\mathcal{T}$. The contradicting trail elements at *dlevel* are collected in *targetLits*. All other contradicting trail elements not in *dlevel* are complemented and stored in $\mathcal{C}$. The *findUIP* procedure (given in Algorithm 12) is used to find the UIP from the trail elements stored in *targetLits*. The backjump level *blevel* is determined from the decision level of the elements of ϕ_g that are not in *targetList*. The elements of $\mathcal{C}$ along with the complement of the UIP forms a conflict clause which is added to the set of abstract transformers $\mathcal{A}$ and finally returned by the procedure Algorithm 11.

After the conflict clause is obtained from the procedure *getConflictClause* in line 2 of Algorithm 10, the procedure *analyzeConflict* undoes all the elements from the trail until *blevel* using the *backjump* procedure and returns *true*. Else if $blevel \leq 0$, the procedure *analyzeConflict* returns *false*.

Abstract UIP Search

Recall from Section 2.2 that an UIP in the implication graph is a vertex v at the current decision level such that every path from the decision variable of the current decision level to the vertices corresponding to the conflicting variable passes through v. An UIP is a single reason that is sufficient to infer the conflict on the largest decision level. Further, the decision variables are also UIPs. Similarly, an abstract UIP is a node in the ACG that must be traversed on every path between the current decision node and the conflict. An abstract UIP cut is necessary to ensure that the learned transformers are asserting after backtracking. An abstract UIP search procedure is presented in Algorithm 12. The algorithm traverses the trail $\mathcal{T}$ starting from the conflict node and computes a cut that suffices to produce a conflict. As before, the trail can seen as representing an ACG that records the sequence of deductions in the domain that are inferred from a decision.

Discussion of Algorithm 12

Algorithm 12 presents the abstract first UIP search procedure $findUIP(targetLits, \mathcal{A}, \mathcal{T}, \mathcal{R})$. The input to the algorithm are the elements of the conflict reason at the current decision

Algorithm 10: Conflict Analysis $analyzeConflict_{H_C}(\mathcal{A}, \mathcal{T}, \mathcal{R})$

input : A set of abstract transformers $\mathcal{A}$, propagation trail $\mathcal{T}$ and reason trail $\mathcal{R}$
output : false if no further backtracking is possible, else true

```
1 blevel ← −1
2 C ← getConflictClause(A, T, R, blevel)
3 if blevel ≤ 0 then
4 |   return false
  /* backjump until blevel                                    */
5 backjump(T, R, blevel)
6 return true
```

level, *targetLits*, the set of abstract transformer, $\mathcal{A}$, the trail $\mathcal{T}$, and reason trail $\mathcal{R}$. The output is the abstract first UIP element.

The *findUIP* algorithm operates on the trail segment containing the elements of the conflict reason at the current decision level. The algorithm uses a *marking* data-structure m that maps indices of $\mathcal{T}$ to a generalized trail element that still preserves the conflict. All entries in the marking array m are initialized to $\top$. The procedure *updateMarking* is called on the elements of the conflict reason that are in the current decision level. The procedure *updateMarking* first decomposes the generalized reason ϕ_g into the set of meet irreducibles Φ. We assume that every element ρ in the decomposition of the generalized reason Φ has a comparable meet irreducible at some index i in $\mathcal{T}$ such that ρ generalizes the corresponding trail element at that index.

UpdateMarking updates the trail marking m with a generalized reason $m[i] = \rho$ where the index i is the earliest trail index, obtained through the procedure *findIndex*, such that the meet irreducible ρ generalizes the trail element $\mathcal{T}[i]$. Note that there may be multiple incomparable generalized reasons for each application of $abdgen^{D}_{\sigma,\phi}$. The abstract conflict analysis procedure heuristically picks one reason among the various incomparable reasons based on the choice of the UIP.

The *while* loop in Algorithm 12 iterates backwards through $\mathcal{T}$ and replaces the trail marking $m[i]$ with the reason ϕ_g that is sufficient to infer $m[i]$ through $\mathcal{R}[i]$. The *UpdateMarking* procedure decomposes the reason ϕ_g and updates the trail markings with the generalized meet irreducibles which overapproximates the corresponding trail element at that position. The procedure *countDeduce* takes an element $\mathcal{R}[i]$ and returns the total number of deductions inferred by the i-th element of the reason trail $\mathcal{R}$. The data structure *dedCount* is used to keep track of this information which maps elements of $\mathcal{R}$ at index i to the total number of propagations made by $\mathcal{R}[i]$ and stores it in $dedCount[i]$. An invariant of the *while* loop of Algorithm 12 is that the meet of all markings in the marking array m after each iteration of

Algorithm 11: *getConflictClause*$(\mathcal{A}, \mathcal{T}, \mathcal{R}, blevel)$

input : A set of abstract transformers $\mathcal{A}$, propagation trail $\mathcal{T}$ and reason trail $\mathcal{R}$, backjump level *blevel*

output : A set $\mathcal{C}$ containing elements of learned clause

$\mathcal{C} \leftarrow \{\}$

/* Store literals of current decision level in *targetLits* */

$targetLits \leftarrow \{\}$

$maxReasonDl \leftarrow -1$

$dlevel \leftarrow$ current decision level

$[\![\sigma]\!]_D \leftarrow \mathcal{R}[\bot]$

/* find the symbols in σ */

$Vars \leftarrow getSymbols([\![\sigma]\!]_D)$

/* find the meet irreducibles that contain elements of *Vars* from the trail segment in *dlevel* */

$f \leftarrow getMatchingMeetIrreducible(\mathcal{T}_{dlevel}, Vars)$

/* find the generalized f */

$v \leftarrow abdgen^{D}_{\sigma,\bot}(f)$

/* get all decisions upto dlevel-1 */

$dec \leftarrow getDecisions(\mathcal{T}, \mathcal{R}, dlevel - 1)$

/* get the initial generalized conflict reason */

$\phi_g \leftarrow v \&\& dec$

/* compute the learned clause and *blevel* */

foreach $\rho \in decomp(\phi_g)$ **do**

  $contrDl \leftarrow findContradiction(\rho, \mathcal{T})$

  if $contrDl = dlevel$ **then**

    $\{targetLits\} \leftarrow \{targetLits\} \cup \{\rho\}$

  else

    if $contrDl > maxReasonDl$ **then**

      $maxReasonDl \leftarrow contrDl$

      $\{\mathcal{C}\} \leftarrow \{\mathcal{C}\} \cup \{\neg(\rho)\}$

end

if $maxReasonDl = -1$ **then**

  $blevel = 0$

else

  $blevel = maxReasonDl$

$uip \leftarrow findUIP(targetLits, \mathcal{A}, \mathcal{T}, \mathcal{R})$

/* Add the uip to the set $\mathcal{C}$ */

$\{\mathcal{C}\} \leftarrow \{\mathcal{C}\} \cup \{\neg uip\}$

/* Add $\mathcal{C}$ to the initial set of transformers $\mathcal{A}$ */

$\mathcal{A} \leftarrow \mathcal{A} \cup (\bigcup_{c \in \mathcal{C}} c)$

/* returns the meet irreducibles of $\mathcal{C}$ */

return $(\bigcup_{c \in \mathcal{C}} c)$

Algorithm 12: Finding the abstract first UIP $findUIP(targetLits, \mathcal{A}, \mathcal{T}, \mathcal{R})$

input : Elements of conflict reason at current decision level *targetLits*, set of abstract transformers $\mathcal{A}$, propagation trail $\mathcal{T}$ and reason trail $\mathcal{R}$
output : Returns abstract first UIP

$m \leftarrow \{1 \rightarrow \top, \ldots, |\mathcal{T}| \rightarrow \top\}$
foreach $\tau \in targetLits$ **do**
  $updateMarking(m, \mathcal{T}, \tau)$
end
$N \leftarrow deepest\ Decision\ level$
$i \leftarrow |\mathcal{T}|$
while $checkUIP(m, \mathcal{T}, N)$ **do**
  $k \leftarrow countDeduce(\mathcal{R}[i])$
  $t \leftarrow countDeduce(\mathcal{R}[i-k])$
  $\phi \leftarrow m[i]$
  if $\phi = \top$ **then**
    continue
  $[\![\sigma]\!]_D \leftarrow \mathcal{R}[i]$
  $a \leftarrow \bigsqcap_{(i-k-t) \leq j < (i-k)} (\mathcal{T}[j])$
  $\phi_g \leftarrow abdgen^D_{\sigma,\phi}(a)$
  $updateMarking(m, \mathcal{T}, \phi_g)$
  $m[i] \leftarrow \top$
  $i \leftarrow i - 1$
end
/* m must contain only one element at this point */
$uip \leftarrow \bigsqcap_{1 \leq i \leq |\mathcal{T}|} m[i]$
return uip

Algorithm 13: $updateMarking(m, \mathcal{T}, \phi_g)$

input : Marking array m, a propagation trail $\mathcal{T}$, and generalized deduction ϕ_g
output : Store ϕ_g in m in the appropriate place

$\Phi \leftarrow decomp(\phi_g)$
foreach $\rho \in \Phi$ **do**
  $j = findIndex(\rho, \mathcal{T})$
  $m[j] = m[j] \sqcap \rho$
end

Algorithm 14: *checkUIP*$(m, \mathcal{T}, N)$

input : Marking array, m, a propagation trail $\mathcal{T}$
output : *TRUE* if there is a UIP in m else false

```
1  count ← 0
2  foreach ρ ∈ m do
3  |  if ρ = ⊤ then
4  |  |  continue
5  |  else
6  |  |  dl ← dLevel(ρ)
7  |  |  count ← count + 1
8  end
9  if count = 1 ∧ dl = N then
10 |  return TRUE
11 else
12 |  return false
```

Algorithm 15: *countDeduce*$([\![\sigma]\!]_D)$

input : an overapproximate deduction transformer $[\![\sigma]\!]_D$
output : total number of deductions from $[\![\sigma]\!]_D$

```
1  c ← dedCount[[σ]_D]
2  return c
```

the loop gives a sufficient reason for conflict. However, to find the first UIP, the *while* loop in Algorithm 12 terminates when there is exactly one element at the deepest decision level in the marking data structure. This check is performed by the procedure *checkUIP*. It is worth noting that a similar learning technique is used by Pseudo-Boolean (PB) solvers [194] for learning general PB constraint which is given by the linear inequality $\sum_{n}^{j=1} a_j.l_j \geq b$, where $j \in \{1,\ldots,n\}$, $a_j \in Z^+$, $b \in Z^+$ and l_j is a literal. Similar to the expressiveness of the learned transformer which contains elements of the underlying abstract domain, the

Algorithm 16: *findIndex*$(\rho, \mathcal{T})$

input : a meet irreducible ρ, a propagation trail $\mathcal{T}$
output : an index for ρ in $\mathcal{T}$ if found, else return -1

```
1  j ← 0
2  foreach a ∈ T[j] do
3  |  if a ⊑ ρ then
4  |  |  return j
5  |  else
6  |  |  j ← j + 1
7  end
8  return −1
```

PB constraints are also more expressive than propositional clause learning since a single PB constraint can represent a large number of propositional clauses. Hence, the potential for pruning from learning PB constraints is much larger than that of propositional clauses. PB solvers use sequence of cutting-plane steps [51, 187] instead of sequence of resolution steps used by propositional SAT solvers, in order to learn PB constraints that are consistent with the PB formula. Similar to the marking-based algorithm in ACDLS, PB solvers also remove one implied variable in each cutting-plane step which finally generates an assertive PB constraint that is unsatisfied under the current assignment. Note that PB solvers also use alternative learning schemes which can learn *propositional clauses* [156] or *cardinality constraints* [194].

7.7.2 Conflict Analysis with an Example

Let us revisit Example 7.7.4. Given $\varphi = (x+y=z) \wedge (x+z=2y) \wedge (z+y>10)$, and a decision $(x \leq 0)$ which leads to the conflict, we show the step-wise execution of Algorithm 12 to compute the UIP over the Interval and the Octagon domains. Figure 7.11 and Figure 7.12 gives an example of abstract UIP search over the Interval and Octagon domains respectively. Figure 7.11 and Figure 7.12 show the content of $\mathcal{T}$, *reason trail* $\mathcal{R}$, ACG, and the marking data-structure m, which stores for each element of $\mathcal{T}$ a generalized reason for conflict. Initially, m maps all elements of the $\mathcal{T}$ to $\top$.

Finding an Abstract UIP in the Interval Domain

Figure 7.11 shows the run of Algorithm 12 for finding a UIP over the Interval domain for $\varphi \mathrel{\hat{=}} x+4=z \wedge x+z=2y \wedge z+y>10$. The ACG in top part of Figure 7.11 records the deductions made following the decision $x \leq 0$. Clearly, the decision leads to a *conflict*, marked by $\bot$.

The algorithm first determines that $\bot$ can still be derived when $z \leq 6 \wedge y \leq 4$. In the first iteration, the algorithm determines that $y \leq 4$ can be dropped from the conflict reason if $x \leq 2$ holds in combination with $z \leq 6$. Finally, a generalized reason for conflict is derived by dropping $z \leq 6$ from the conflict reason since $x \leq 2$ is sufficient to derive the conflict. We now show the content of the marking data-structure for each iteration of the Algorithm 12.

1. Before entering the for loop:
   $m \leftarrow \{1 \rightarrow \top, \ldots, |\mathcal{T}| \rightarrow \top\}$
   $abdgen^{Itvs}_{z+y>10,\bot}(x \leq 0 \wedge z \leq 4 \wedge y \leq 2) = \{z \leq 6, y \leq 4\}$
   Entries of marking array m: $m[1] = \top, m[2] = z \leq 6, m[3] = y \leq 4$

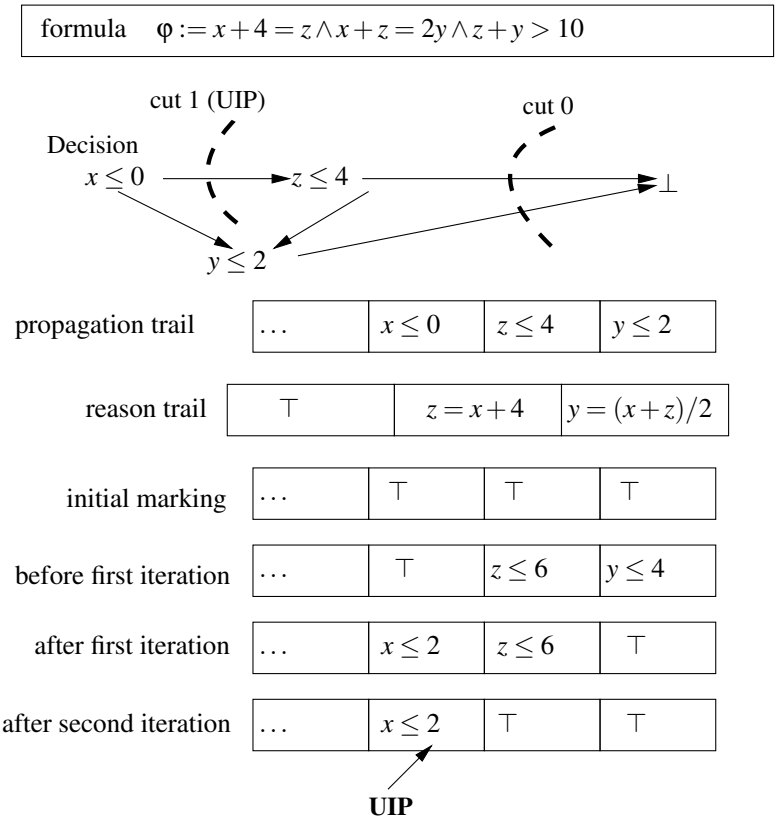


Figure 7.11: Finding an abstract UIP in the Interval domain

2. Since $|\mathcal{T}|$ = 3, for i=3, $abdgen^{Itvs}_{y=(x+z)/2,y\leq 4}(x \leq 0 \wedge z \leq 4) = \{x \leq 2, z \leq 6\}$
   Entries of marking array m: $m[1] = x \leq 2, m[2] = z \leq 6, m[3] = \top$

3. For i=2, $abdgen^{Itvs}_{z=x+4,z\leq 6}(x \leq 2) = \{x \leq 2\}$
   Entries of marking array m: $m[1] = x \leq 2, m[2] = \top, m[3] = \top$

Finding an Abstract UIP in the Octagon Domain

We now demonstrate abstract UIP computation over the Octagon domain for Example 7.7.4. The abstract conflict graph in Figure 7.12 records the deductions following decision $x \leq 0$. Clearly, the decision leads to a *conflict*, marked by $\bot$. The dotted arrows in ACG of Figure 7.12 denotes the set of deductions which are made from the same abstract transformer. Algorithm 12 finds the first UIP after an application of $abdgen^{D}_{\sigma,\phi}$. Similar to abstract UIP search in the Interval domain, the marking data-structure is used to store a generalized reason corresponding to each element of $\mathcal{T}$. The content of the marking data-structure at each iteration of Algorithm 12 is shown below. Although Algorithm 12 terminates after returning the first UIP, Figure 7.12 shows the computation of last UIP as well.

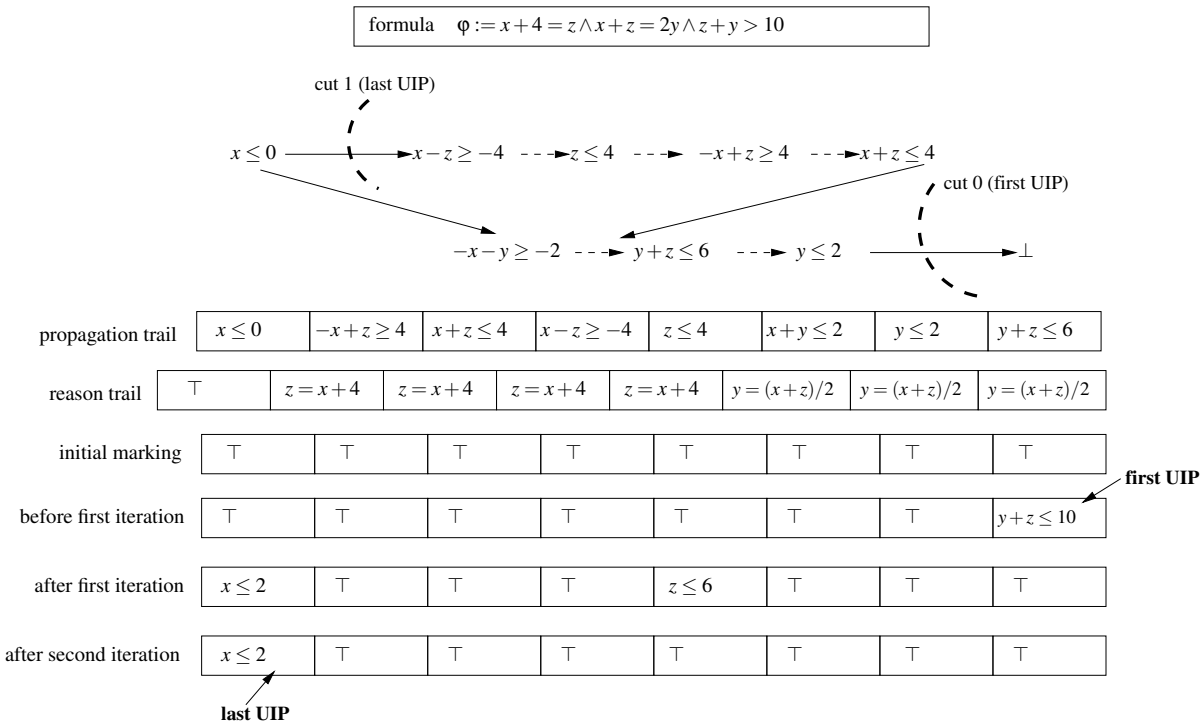


Figure 7.12: Finding an abstract UIP in the Octagon domain

1. Before entering the for loop:
   $m \leftarrow \{1 \rightarrow \top, \ldots, |\mathcal{T}| \rightarrow \top\}$
   $abdgen^{Octs}_{z+y>10,\bot}(abs(\mathcal{T})) = \{y+z \leq 10\}$
   Entries of marking array, m: $m[1] = \top, m[2] = \top, m[3] = \top, m[4] = \top, m[5] = \top, m[6] = \top, m[7] = \top, m[8] = y+z \leq 10$

Finding All UIPs

We demonstrate the computation of all UIPs at the current decision level, similar to GRASP [196]. However, note that ACDLS currently support only the first UIP and last UIP based learning. Recall that Algorithm 12 stops when there is exactly one element in the marking data-structure at the deepest decision level. However, in order to compute all UIPs, the trail segment corresponding to the deepest decision level has to be completely traversed. Below we show the computation of all UIPs for the ACG of Figure 7.12 over the Octagon domain. For this example, there are only two UIPs – the first UIP and the last UIP.

1. Before entering the for loop:
   $m \leftarrow \{1 \rightarrow \top, \ldots, |\mathcal{T}| \rightarrow \top\}$
   $abdgen^{Octs}_{z+y>10,\bot}(abs(\mathcal{T})) = \{y+z \leq 10\}$

Entries of marking array m: $m[1] = \top, m[2] = \top, m[3] = \top, m[4] = \top, m[5] = \top, m[6] = \top, m[7] = \top, m[8] = y + z \leq 10$

2. *For i=8*,
   $abdgen^{Octs}_{y=(x+z)/2, y+z \leq 10}(x \leq 0 \wedge -x+z \geq 4 \wedge x+z \leq 4 \wedge x-z \geq -4 \wedge z \leq 4) = \{x \leq 2, z \leq 6\}$
   Entries of marking array m: $m[1] = x \leq 2, m[2] = \top, m[3] = \top, m[4] = \top, m[5] = z \leq 6, m[6] = \top, m[7] = \top, m[8] = \top$

3. *For i=7*, continue. *For i=6*, continue.

4. *For i=5*, $abdgen^{Octs}_{z=x+4, z \leq 6}(x \leq 2) = \{x \leq 2\}$
   Entries of marking array m: $m[1] = x \leq 2, m[2] = \top, m[3] = \top, m[4] = \top, m[5] = \top, m[6] = \top, m[7] = \top, m[8] = \top$

5. *For i=4*, continue. *For i=3*, continue. *For i=2*, continue. *For i=1*, continue.

Note that the trail $\mathcal{T}$ obtained from the Interval and Octagon analyses are different. Hence, the structure of the ACG in Figure 7.11 and Figure 7.12 are different for the same formula φ, which leads to different UIPs. Note that the last UIPs in Figure 7.11 and Figure 7.12 are the same since they are obtained by generalizing the decision node. However, the first UIPs differ between Figure 7.11 and Figure 7.12.

Finding all UIPs in ACG of Figure 7.12 requires first determining a generalized conflict reason, $y + z \leq 10$, which is sufficient to derive $\bot$. The procedure then determines that the meet irreducible $y + z \leq 10$ can be dropped from the conflict reason if $z \leq 6$ in combination with $x \leq 2$ holds. Note that the rationale behind dropping the octagonal constraint $y + z \leq 10$ from the conflict reason is that it may give opportunity to learn a sufficiently generalized conflict reason. Also, we avoid learning constraints that are directly implied by the abstract transformers (for example, $y + z > 10$ in this case). Finally, a generalized reason for the conflict is derived by dropping $z \leq 6$ from the conflict reason. The meet irreducible $x \leq 2$ is sufficient to derive the conflict. Table 7.13 gives the candidate cuts and UIPs for the ACGs in Figure 7.11 and Figure 7.12. The first three rows denote the cuts obtained from the ACG in Figure 7.11 over the Interval domain. The corresponding generalizations of these cuts are shown in the third column. Note that the cut in the third row is a UIP.

The last three rows in Table 7.13 gives the candidate cuts and UIPs for the ACG in Figure 7.12 over the Octagon domain. In contrast to a single UIP obtained in Figure 7.11, there are two different UIPs for the ACG corresponding to Figure 7.12.

Domain	Cut	Generalized Cut
Cuts in Figure 7.11		
Interval	$\{y \leq 2, z \leq 4\}$	$\{y \leq 4, z \leq 6\}$
Interval	$\{x \leq 0, z \leq 4\}$	$\{x \leq 2, z \leq 6\}$
Interval (UIP)	$\{x \leq 0\}$	$\{x \leq 2\}$
Cuts in Figure 7.12		
Octagon	$\{x \leq 0, z \leq 4\}$	$\{x \leq 2, z \leq 6\}$
Octagon (UIP)	$\{x \leq 0\}$	$\{x \leq 2\}$
Octagon (UIP)	$\{y+z \leq 6\}$	$\{y+z \leq 10\}$

Figure 7.13: Candidate cuts and UIPs for ACGs in Figure 7.11 and Figure 7.12 over the Interval and Octagon domains respectively

Learning in the Template Polyhedra Domain

Learning in a propositional solver yields an asserting clause [196, 226] that expresses the negation of the conflict reasons. Recall that Section 6.9.3 presents a lattice-theoretic generalization of the *unit rule* called *abstract unit transformer*, $AUnit_C(a)$, for the abstract value a and the set of meet irreducibles C such that $\bigsqcap C$ does not satisfy the SSA formula ϕ. We add $AUnit_C(a)$ to the set of abstract transformers $\mathcal{A}$. The abstract unit transformer is a generalization of the propositional unit rule for numerical domains. We now give an example to illustrate the application of the $AUnit_C(a)$ transformer. Note that the rules specified in Example 7.7.5 refers to the corresponding rules defined in Section 6.9.3.

Example 7.7.5. An example of $AUnit_C(a)$ for $C = \{x \geq 2, x \leq 5, y \leq 7\}$ is given below.

Rule 1: For $a = (x \geq 3 \wedge x \leq 4 \wedge y \geq 5 \wedge y \leq 6)$, $AUnit_C(a) = \bot$, since $a \sqsubseteq \bigsqcap C$.

Rule 2: For $a = (x \geq 3 \wedge x \leq 4)$, $AUnit_C(a) = (y \geq 8)$, since $a \sqsubseteq (2 \leq x \leq 5)$.

Rule 3: For $a = (x \geq 1 \wedge y \leq 10)$, $AUnit_C(a) = \top$.

Implementation of the Abstract Unit Transformer

We use the notion of subsumption check for implementing $AUnit_C(a)$. The subsumption check is also commonly known as *Inclusion test* which is available in the APRON library. The subsumption check in ACDLS is performed by the abstract domain operator. Note that the meet irreducibles of C are stored in a C++ vector STL and each C is stored in a learned transformer database which is implemented by C++ stack STL.

Given two abstract values, subsumption check [49,130,185] is used to determine whether one abstract value is entailed by the other. The subsumption check returns true in case all the concrete values represented by one abstract value are also represented by the other

abstract value. For example, given two meet irreducibles $x \leq 10$ and $x \leq 5$, we say that $x \leq 5$ is subsumed by $x \leq 10$ since the set of concrete values represented by the interval $x \leq 5$ are included by the interval $x \leq 10$. Subsumption check between two polyhedra $P1$ and $P2$, denoted by $P1 \subseteq P2$, is implemented using linear programming, which checks whether each inequality in $P2$ is entailed by $P1$. That is, for each $\sum_i a_i x_i \leq b$ in $P2$, the value $\tau = \texttt{max} \sum_i a_i x_i$ subject to $P1$ is computed. If $\tau > b$, then the subsumption does not hold.

For the abstract value a and the set of meet irreducibles C such that $\bigsqcap C$ does not satisfy ϕ, the result of $AUnit_C(a)$ is determined by the comparison given by $a \sqsubseteq \bigsqcap C$, which checks whether each meet irreducible in C is entailed by a. $AUnit_C(a)$ returns $\bot$ if each meet irreducible in C is indeed entailed by a. However, if all meet irreducibles in C but one is entailed by a, then C is unit. In this case, $AUnit_C(a)$ infers a valid meet irreducible which is just the complement of the meet irreducible in C that is not entailed by a. On the other hand, if none of the meet irreducibles in a entails the meet irreducibles in C, then $AUnit_C(a)$ returns $\top$ which implies that the learned transformer is not *asserting* after backtracking.

7.8 Experimental Results

We have implemented ACDLS for automatic bounded safety verification of the software netlist designs (in C) generated from the hardware RTL and other complex softwares. ACDLS is implemented in C++ on top of the CPROVER [3] framework as an extension of 2LS [190] and consists of around 9 KLOC. The template polyhedra domain is implemented in C++ in 10 KLOC. Templates can be intervals, octagons, zones, equalities, or restricted polyhedra. Our domain handles all C operators, including bit-wise ones, and supports precise complementation of meet irreducibles, which is necessary for conflict-driven learning.

7.8.1 Benchmark and Tool Distribution

The source code of ACDLS is available at `http://www.cprover.org/acdcl/`. The installation process of ACDLS is described in the above website.

Command Line Options

The following command is used for running ACDLS.

```
2ls --acdl <file-name> --decision <D> --propagate <P> --learning <L> --<domain> --inline
```

ACDLS expects the input file to be a C program in .c format. The decision heuristic `D` can be `ordered`, `longest-range`, `berkmin` or `random`. The propagation heuristics

[3] http://www.cprover.org/

`P` can be `forward`, `backward` or `chaotic`. The learning heuristic `L` is `first-uip`. However, if no learning is used, then this option is redundant and can be ignored. The abstract domain `domain` can be `intervals`, `octagons` or `equalities`. For example, assuming that the input file name is `main.c`, the command for running ACDLS is given as follows.

```
2ls --acdl main.c --decision ordered --propagate chaotic --learning first-uip --inline
```

Benchmarks

A collection of benchmarks is available at `http://www.cprover.org/acdcl/`. The benchmarks are derived from the various sources – (1) Software netlist (in C) of the hardware circuits (in Verilog) auto-generated by v2c [175] from VIS Verilog models and opencores.org; (2) controller code with varying loop bounds auto-generated from Simulink model; (3) the bit-vector regression category in SV-COMP'16; (4) hand-crafted instances that implement programs with complex control logic and contains relational properties. We classify our benchmarks into three separate categories. We label the software netlist derived from Verilog RTL design in *Verilog-C* category; auto-generated Controller code and control-intensive benchmarks in *Control-Flow* category; bit-vector regression benchmarks from SV-COMP'16 in *Bit-vector* category. The total number of benchmarks in *Verilog-C* category is 17. Out of these 17 benchmarks, 10 are safe and the remaining 7 are unsafe. The Verilog-C category include an implementation of an instruction buffer logic, FIFO arbiter, traffic light controller, cache coherence protocol and Dekker's mutual exclusion algorithm among others. The largest benchmark in this category is the cache coherence protocol which consists of 890 LOC and the smallest benchmark is TicTacToe with 67 LOC. The Control-Flow category contains 55 benchmarks. Out of 55 benchmarks in this category, 35 are safe and 20 are unsafe. The Bit-vector category contains a total of 13 benchmarks, out of which 6 are safe and the remaining 7 are unsafe benchmarks.

7.8.2 Discussion of the Results

We verify a total of 85 ANSI-C benchmarks. All the programs have bounded loops which are completely unrolled before analysis. We compare ACDLS with the state-of-the-art SAT-based bounded model checker CBMC [4] (version 5.5) and a commercial abstract interpretation tool, Astrée [5] (version 14.10). CBMC uses MiniSAT 2.2.1 in the backend. Astrée uses a range of abstract domains, which includes interval, bit-field, congruence, trace

[4] http://www.cprover.org/cbmc/

[5] https://www.absint.com/astree/index.htm

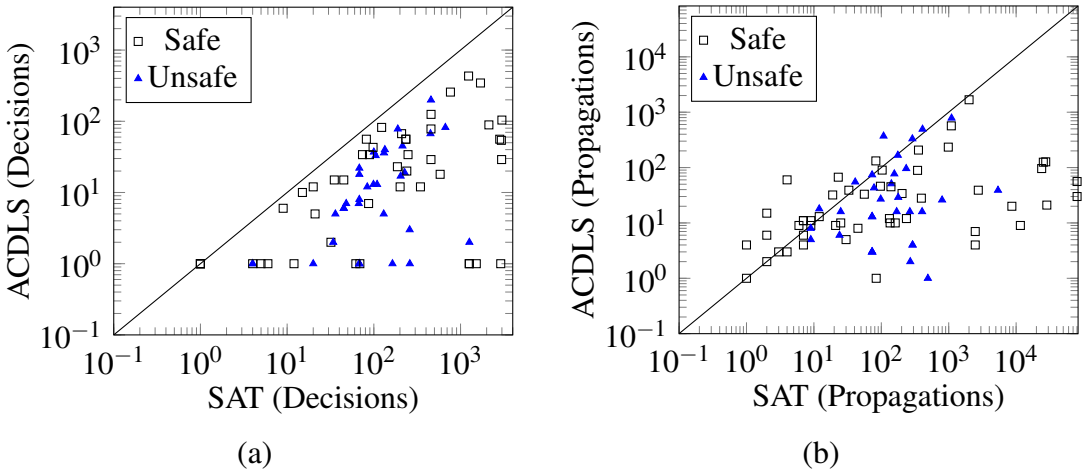


Figure 7.14: Comparing SAT-based BMC and ACDLS: number of decisions and propagations

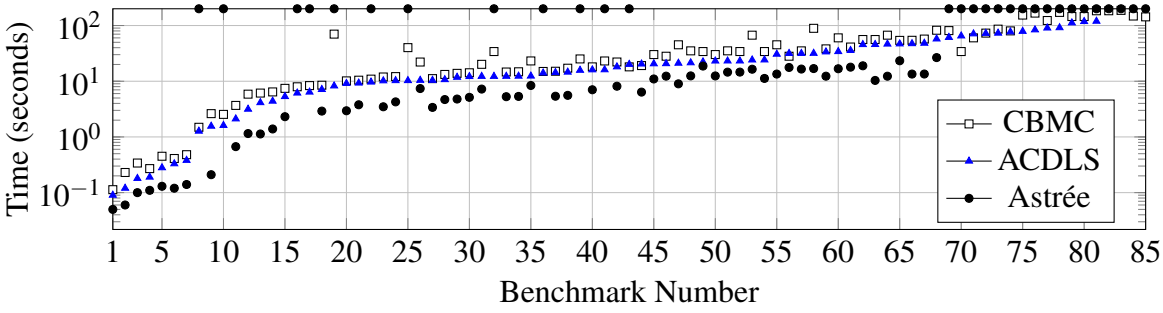


Figure 7.15: Runtime comparison between CBMC, Astrée and ACDLS

partitioning, and relational domains (octagons, polyhedra, zones, equalities, filter). To enable fair comparison using Astrée, all bounded loops in the program are completely unwound up to the maximum bound before passing to Astrée. This prevents Astrée from widening loops. ACDLS is instantiated with a product of the Booleans and the Interval or Octagon domains. ACDLS is also configured with a decision heuristic (ordered, random, activity-based), propagation heuristic (forward, backward and multi-way), and conflict-analysis (learning UIP, DPLL-style). The timeout for verification of each property is set to 200 seconds.

ACDLS versus CBMC

Figure 7.14 presents a comparison between CBMC and ACDLS. The SAT solver statistics for CBMC in Figure 7.14 is obtained from MiniSAT 2.2.1. In Figure 7.14(a), the x-axis gives the number of decisions in CBMC using MiniSAT and the y-axis gives the number of decision in ACDLS. In Figure 7.14(b), the x-axis gives the number of propagations in CBMC using MiniSAT and the y-axis gives the number of propagations in ACDLS.

Figure 7.14(a) shows that the SAT-based analysis makes significantly more decisions than ACDLS for all the benchmarks. In Figure 7.14(b), SAT-based analysis makes more deductions than ACDLS on average. The points on the extreme right below the diagonal in Figure 7.14 show maximal propagations for the SAT-based analysis. These benchmarks exhibit relational behaviour and are solved by the Octagon domain in ACDLS. Out of 85 benchmarks, SAT-based analysis could prove only 26 benchmarks without any restarts. The solver was restarted in the other 59 cases to avoid spending too much time in "hopeless" branches. By contrast, ACDLS solved 81 benchmarks but timeout in 4 cases.

The performance comparison between ACDLS and CBMC can be explained by two factors - runtime performance and efficiency. The runtime comparison between ACDLS and CBMC is shown in Figure 7.15. The *x*-axis of Figure 7.15 gives the Benchmark number and the *y*-axis gives the time taken by ACDLS and CBMC. The results show that ACDLS is on average 1.5X faster than CBMC. The superior runtime performance of ACDLS is attributed to the program-specific decision heuristics, which exploit the high-level structure of the software netlist, combined with the precise deductions from the chaotic propagation heuristic and the expressive learned transformers that contains elements from the underlying abstract domains.

Figure 7.15 shows that the runtimes of CBMC is very close to ACDLS for 75 out of 85 benchmarks. That is, the time taken by the MiniSAT solver to reason about the bit-blasted formulas is marginally higher than the time taken by ACDLS to solve the equivalent non-bit-blasted SSA formulas derived from the software netlist designs of the RTL. The consistent runtimes of CBMC is due to the significant advances of the underlying SAT solving technology which have been heavily engineered for its performance over the last 2 decades. In particular, SAT solvers use lazy data structures, watched literals, restart strategies and problem specific decision, propagation and conflict heuristics for its optimized performance. However, the real benefit of ACDLS can be seen from the fact that it is significantly more efficient (in terms of the total number of decisions, propagations and learning) than the SAT-based analysis in CBMC (see Figure 7.16). This clearly demonstrates the benefit of high-level reasoning using abstract CDCL for the purposes of RTL verification over the conventional SAT-based bit-blasting approach. Furthermore, we believe that the continued engineering efforts to improve the performance of ACDLS can reap the benefits of the efficiency gains for more superior runtime performance compared to the conventional bit-blasting approach.

ACDLS versus Astrée

Table 7.3 gives a detailed comparison between Astrée and ACDLS. Columns 1–5 in Table 7.3 gives the name of the tool, the benchmark category, the total number of instances

Verifier	Category	#Proved (safe/unsafe)	#Inconclusive	#False Positives
Astrée	Bit-vector	5/7	0	1
ACDLS		6/7	0	0
Astrée	Control-Flow	24/9	0	22
ACDLS		35/17	3	0
Astrée	Verilog-C	4/7	0	6
ACDLS		9/7	1	0

Table 7.3: Astrée versus ACDLS

proved safe or unsafe (labelled as safe/unsafe), the total number of inconclusive benchmarks and the total number of false positives per category. All times in Table 7.3 are given in seconds. To enable precise analysis with Astrée, we manually instrument the benchmarks with partition directives `__ASTREE_partition_control` at various control-flow joins. These directives provide external hints to Astrée to guide its internal trace partitioning domain. Table 7.3 shows that ACDLS solved 25 more benchmarks than Astrée. The total number of inconclusive results in ACDLS is 4, which are all timeouts. The analysis using ACDLS produces correct results for 81 benchmarks without any false positives. By contrast, Astrée reports a total of 29 false positives among 85 benchmarks and correctly solves 56 benchmarks. Clearly, ACDLS is sufficiently more precise than Astrée. Figure 7.15 demonstrates that Astrée is 2X faster than ACDLS for 65% of the cases (56 out of 85); but the analysis using Astrée shows a high degree of imprecision (marked as timeout in Figure 7.15). Furthermore, Astrée required manual guidance to solve 76% of the correctly solved benchmarks (43 out of 56). By contrast, analysis using ACDLS is completely automatic.

7.8.3 Decision Heuristics in ACDLS

ACDLS support various decision heuristics – *ordered*, *longest-range*, *random*, and the *activity based* decision heuristic. Recall from Section 7.6.5 that a decision in ACDLS returns a meet irreducible expressible by the underlying abstract domain. The *ordered* decision heuristic creates an ordering among the SSA variables in the $\widehat{\mathcal{SA}_G}$ lattice. It first makes decisions on conditional variables (variables that appear in conditional branches and marked with prefix `cond`), before choosing numerical variables. Recall that an element of $\widehat{\mathcal{SA}_G}$ is given by $\widehat{\mathcal{SA}_G} \hat{=} C\vec{x} \leq \vec{d}$. The *longest-range* heuristic simply keeps track of the bounds d_l, d_u of matching template rows, which are row vectors $\vec{c}$, $\vec{c}^{\dagger}$ such that $\vec{c} = -\vec{c}^{\dagger}$. For $d_l \leq \vec{c}\vec{x} \leq d_u$, the heuristic picks the one with the longest range $d_u - d_l$, and returns the meet irreducible $\vec{c}\vec{x} \leq \lfloor \frac{d_l + d_u}{2} \rfloor$ or its complement. The *random* decision heuristic arbitrarily picks an SSA variable with the given bounds d_l, d_u and returns the meet irreducible $\vec{c}\vec{x} \geq \lfloor \frac{d_l + d_u}{2} \rfloor$. A biased binary search solver is used to first select the bounds from the higher range. Only

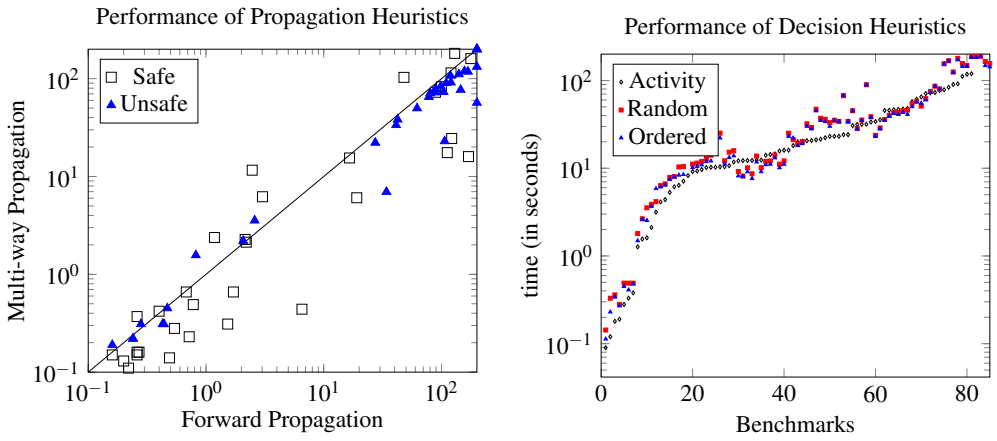


Figure 7.16: Effect of propagation heuristics and decision heuristics in ACDLS

when the higher range is exhausted and no model has been found, the bounds are selected from the lower range. The *activity-based* decision heuristic keeps track of the activity of variables that participate in the conflict transformers. Based on the most active variable, ranges are split similar to the *longest-range* heuristic.

7.8.4 Effect of Decisions, Propagations and Learning in ACDLS

Propagation Strategy. Figure 7.16(a) presents a comparison between the *forward* and *multi-way* propagation strategy in ACDLS. Note that the multi-way strategy works with the `chaotic` propagation heuristic. The x-axis of Figure 7.16(a) gives the runtime for forward propagation and the y-axis gives the runtime for the multi-way propagation. The choice of the propagation strategy influences the total number of decisions and transformer learning iterations. Hence, the propagation strategy has a significant influence on the runtime, as can be seen in the Figure 7.16(a). Overall, the multi-way strategy performs best and is superior than the forward propagation for most of the cases. Owing to the precise deductions from the multi-way strategy, it converges faster on our benchmarks compared to the forward propagation strategy. However, both heuristics timeout on 4 out of 85 benchmarks. The *backward* propagation strategy shows worst performance on our benchmarks and reports large number of timeouts. So, we do not report the performance of the *backward* propagation strategy.

Decision Heuristics Figure 7.16(b) gives the performance of different decision heuristics in ACDLS. The x-axis of Figure 7.16(b) gives the total number of benchmarks and the y-axis gives the runtime for each decision heuristic. Note that the runtimes for all decision heuristics

are obtained in combination with the multi-way propagation strategy since the multi-way propagation performs best on our benchmarks. The runtimes of different decision heuristics are very similar, but we can still discern some key characteristics of these heuristics. The runtimes for the random decision heuristic are marginally higher than the ordered heuristic and the activity-based heuristic. The activity based decision heuristic performs consistently well for most safe benchmarks and all bit-vector category benchmarks. By contrast, the ordered decision heuristic performs better for programs with conditional branches since it prioritizes decisions on variables that appear in conditionals. The ordered decision heuristic gives an effect of trace partitioning since it partitions the traces of the program by making decisions on conditional branches. On some unsafe programs, we observe that the ordered decision heuristic finds a bug much faster than the other heuristics by choosing those conditional branches along the path that actually leads to the bug. This suggests that domain-specific decision heuristics are important for ACDLS.

Learning Learning has a significant influence on the runtime of ACDLS. We compare the first UIP-based learning technique with an analysis that performs classical DPLL-style analysis. The effect of UIP computation allows ACDLS to backtrack non-chronologically and guide the model search with the learned transformer. Classical DPLL-style analysis exhibits case-enumeration behaviour and could not finish within the time bound for 20% of our benchmarks.

To summarize, our experimental evaluation suggests that ACDLS is more efficient than CBMC with MiniSAT solver and sufficiently more precise than the commercial abstract interpreter Astrée.

Optimizations in ACDLS

We discuss the various optimization strategies implemented in ACDLS for bounded safety verification.

1. ACDLS implements the best abstract transformer using symbolic formulas over the concrete domain for bit-precise reasoning of the complex bit-manipulating RTL operators such as concatenation, bit-extraction, part-select operators and others. The best abstract transformer is implemented using concretisation and abstraction functions. The concretization function maps an abstract value A to a logical formula ϕ, such that ϕ and A represent the same set of concrete values. The concrete domain enables symbolic representation and manipulation of the concrete values through operations on the formula ϕ. The abstraction function then computes the most precise abstract value that overapproximates the result of the operations on ϕ.

2. ACDLS intelligently exploits the high-level program structure for efficiently detecting counterexamples using a specialised decision heuristic which makes decision on the boolean expression that appears in the conditional branches. Thus, when a particular conditional branch is chosen, then the path corresponding to the selected branch becomes feasible and all other paths are marked as infeasible. The subsequent decisions in ACDLS are restricted to the branches that only appear in the feasible path, which further guides the model search to traverse deep along a particular path of the design similar to the path-wise symbolic execution. This might lead to faster counterexample detection if the bug is indeed in the currently explored path.

3. The generalisation of the first UIP learning over template polyhedra domain gives an expressive learned transformer that contains elements from the underlying abstract domains.

4. The multi-way transformer provides more precise deductions than the deductions obtained from the greatest fixed point of the forward transformer. Hence, model search using forward propagation requires significantly more decisions and learning iterations compared to the model search using multi-way transformer.

5. The lazy closure computation using subdomain construction provides an efficient way for computing deductions in relational domains.

6. Decision in ACDLS picks meet irreducibles from the abstract domains. Hence, the expressivity of the decision depends on the expressivity of the underlying abstract domain, for example decisions in Octagon domain may of the form $x + y \leq 10$, or decision in Equality domain may be of the form $x == y$.

7. ACDLS performs property-driven syntactic slicing for removing the program statements that are not relevant to the properties under verification. ACDLS also implicitly performs program and property driven trace partitioning for automatically refining the precision of the analysis.

7.9 ACDLS versus Classical Abstract Interpretation

In this section, we discuss various limitations of the conventional abstract interpretation using Astrée for precise safety verification and how ACDLS can overcome these limitations. The key insight is that ACDLS does not require expensive abstract domains such as the disjunctive domain or trace partitioning domain to overcome these limitations. Rather, the

automatic transformer refinement procedure in ACDLS via learning allows it to recover from imprecision and perform precise verification over a non-distributive abstract domain.

In Section 4.7.3, we discussed some of the limitations of Astrée to verify the software netlist designs generated from the RTL. We observed that manual partitioning of program traces or use of expensive abstract domains can recover some precision. However, it is extremely challenging to discover the necessary partition of the program traces that are precise enough to prove the property, since that requires significant domain knowledge. Additionally, the precise reasoning of the complex bit-wise operations used by the software netlist designs either requires suitable abstract domains that can keep track of relationship between the bits of a variable or precise implementation of the abstract transformers. Astrée does not implement such an abstract domain and hence the analysis often produces imprecise results.

Here, we discuss two key advantages that ACDLS offers over Astrée - 1) ACDLS performs automatic program and property driven trace partitioning, while Astrée often requires manual partitioning, and 2) Analysis using ACDLS uses best abstract transformers for bit-precise reasoning, while Astré often employs specialized abstract domains for such analysis.

Program and Property Driven Trace Partitioning

ACDLS performs automatic program and property driven trace partitioning. This is illustrated with an example in Figure 7.17. Consider the simple program *P* on the left of Figure 7.17. A simple forward Interval analysis cannot prove safety of *P* due to the control-flow join following the `if-else` branch. However, the analysis using ACDLS makes a decision on the variable $\langle$ `y`:[$-$inf, 3] $\rangle$ which implicitly constructs a trace partitioning, as shown on the right-hand side of Figure 7.17. Note that the program on the right is used only for illustrating the effects of trace partitioning in ACDLS. Interval analysis using the above decision immediately lead to safety. At this point, the analysis backtracks, discarding all propagations which lead to conflict, and learns that $\langle$ `y`: [4, $+$inf] $\rangle$. Interval analysis also proves that *P* is safe with the learned transformer. The analysis cannot backtrack further and therefore terminates, proving that the program is safe. This example illustrates that ACDLS automatically performs program and property-driven trace partitioning. The partition is program-dependent because if the branch condition in *P* was `(y<10)`, then ACDLS would have generated a different partition, $\langle$ `y`:[$-\infty$, 9], `y`:[10, $+\infty$] $\rangle$. The partition is property-dependent because if the assertion was `assert(x < 1)`, then no splitting would have been needed to prove safety.

C program	Partitioned Program
`void foo(int x, int y)` `{` `  if(y < 4)` `  x = 1;` `  else` `  x = -1;` `  assert(x != 0);` `}`	`void foo_partitioned()` `{` `  if(y < 4) {` `  foo(x, y);` `  assert(x != 0);` `  }` `  else {` `  foo(x, y);` `  assert(x != 0);` `  }` `}`

Figure 7.17: C Program and its corresponding partitions

Bit-precise Reasoning in ACDLS

Recall that the software netlist designs contain complex bit-wise operations that model RTL operations such as concatenation, bit-extraction, part-select operator and others. A bit-precise analysis of the software netlist design warrants special abstract domains that can keep track of the relationship between the bits of a variable. In Section 4.7.3, we discuss that bit-wise operations are not common in conventional softwares. Hence, software verification tools such as Astrée do not employ such abstract domains.

On the other hand, ACDLS performs automatic bit-precise analysis without employing special abstract domains for such purpose. Instead, ACDLS implements the best abstract transformer using decision procedure (see Section 7.6.2) to precisely reason about the bit-manipulating operations of the software netlist designs. We illustrate this with an example.

Verilog	Software netlist
`reg [7:0] out, tmp;` `wire [7:0] b = 5;` `wire [7:0] d = 1;` `always @(posedge clk)` `begin` `  out[b] = tmp[d];` `end`	`unsigned char b=5, d=1;` `unsigned char tmp, out;` `unsigned width_of_out = 8;` `unsigned width_of_tmp = 8;` `out = (out & (unsigned char)` `((pow(2,width_of_out)-1) - pow(2,b))) |` `(((tmp & (~((unsigned char)` `(pow(2,width_of_tmp)-1 - pow(2,d))))) >> d) << b);`

Figure 7.18: Bit-precise Reasoning in ACDLS

Figure 7.18 gives a Verilog code snippet which performs a single-bit assignment from one RTL register to another register, which is given by the Verilog statement, `out[b]=tmp[d];`. The equivalent software netlist is shown on the right side of Figure 7.18. The software netlist uses complex bit-wise operations such as bitwise AND ($\&$), bitwise OR ($|$), bitwise NOT ($\sim$), left-shift ($<<$), right-shift ($>>$), and power (*pow*) operators to model the Verilog assignment statement, the result of which is assigned to the variable `out`. A bit-precise analysis of the software netlist in Figure 7.18 using techniques such as abstract interpretation

would require an abstract domain that can keep track of the relationship between the specific bits of the variables `b, d, tmp` and `out`. However, ACDLS analyzes the software netlist precisely by synthesizing the best abstract transformers for each operator. That is, given an interval representation for each variable in the program, ACDLS analyzes each statement of the program in the concrete domain which represents the concretization of each variable as logical formulas.

To illustrate this idea, let us consider an expression $g \circ f$ which is the composition of two abstract transformers f and g that are defined over the Interval domain. For the program in Figure 7.18, f and g could be shift operator, bitwise AND/OR, or other operators. The best abstract transformers for operators f and g are given by $\alpha_{Itvs} \circ f \circ \gamma_{Itvs}$ and $\alpha_{Itvs} \circ g \circ \gamma_{Itvs}$, respectively. Note that γ_{Itvs} and α_{Itvs} are the concretization and abstraction functions for the Interval domain. ACDLS computes the transformer, $\alpha_{Itvs} \circ g \circ f \circ \gamma_{Itvs}$. An alternative option is to compute the transformer $\alpha_{Itvs} \circ g \circ \gamma_{Itvs} \circ \alpha_{Itvs} \circ f \circ \gamma_{Itvs}$. However, the later is less precise than the former since the composition $\gamma_{Itvs} \circ \alpha_{Itvs}$ may lead to loss in precision.

We illustrate the bit-precise reasoning in ACDLS with the software netlist design of Figure 7.18. Assume `tmp = 2` and `out = 4`, the assignment statement `out[5] = tmp[1];` gives `out = 36` in the Verilog as well as in the software netlist design. The synthesis of the best abstract transformer can be explained as follows. Consider the sub-expression in the software netlist of Figure 7.18 which is given by the following formula, `((tmp & ~ ((pow(2,width_of_tmp)-1 - pow(2,d))))>>d);`. For the purpose of illustration, we ignore the `unsigned char` cast operator shown in Figure 7.18. Let transformer f be the bitwise AND operator which is given by f(`tmp`, r) = `tmp & r`, where r = ~ `((pow(2,width_of_tmp)-1 - pow(2,d)));`. Let transformer g be the right shift operator such that $g(f$(`tmp`, r), d) = f(`tmp`, r) `>> d`. Given that `width_of_tmp=8`, `tmp=2`, and `d=1`, the result of f can be computed symbolically which is represented by the logical formula, `(2& ~ (255 - 2))`, in the concrete domain. If the variable `d` was symbolic, then $g \circ f$ would have been represented symbolically by the logical formula `((2& ~ (255 - pow(2,d))) >> d)`. For `d=1`, the result of $g \circ f$ is given by the logical formula, `((2& ~ (255 - 2)) >> 1)`, in the concrete domain. Applying Algorithm 9, the precise abstract representation of the above logical formula `((2& ~ (255 - 2)) >> 1)` in the Interval domain is given by the interval `[1,1]`. Thus, ACDLS performs automatic bit-precise reasoning of complex bit-manipulating statements by synthesizing the best abstract transformer for each operator in the concrete domain and then abstracting the result of the transformer over a given abstract domain.

7.10 Conclusions

We present an algorithmic framework and a practical implementation for instantiating ACDLS over the template polyhedra abstract domains for precise safety verification. We present an *abstract model search* procedure that uses parameterized abstract transformer to flexibly control the precision and efficiency of the deductions in the template polyhedra abstract domain for searching counterexamples. The abstract transformers uses a symbolic implementation of the abstraction and concretization functions to precisely reason about the complex bit-wise operations originating from the software netlist designs generated from the RTL circuits. The underlying expressivity of the abstract domain helps our decision heuristics to exploit the high-level structure of the program for making effective decisions.

The *abstract conflict analysis* procedure is used to analyze the reason for the safe or invalid traces from the partial safety proof. The conflict analysis procedure learns new abstract transformers with the UIP over the given template instance of the template polyhedra abstract domain, which subsequently guides the counterexample search away from the conflicting region. We show that learning in ACDLS performs transformer refinement that automatically refines the precision of the analysis. We embody these techniques in a tool called ACDLS for automatic bounded safety verification of the software netlist designs generated from the RTL and other complex softwares. Experimental evaluation over a range of benchmarks show that ACDLS is 1.5× faster than CBMC with MiniSAT solver. Moreover, the analysis using ACDLS is more efficient than MiniSAT in the number of decisions, propagations and learning. We also present a comparison of ACDLS with abstract interpretation using Astrée. Experiments have shown that Astrée correctly solves 56 benchmarks but reports a total of 29 false alarms among 85 benchmarks. By contrast, ACDLS is sufficiently more precise and solves 25 more benchmarks than Astrée. Similar benefits are also seen for SAT-based BMC over Astrée. Furthermore, the analysis using ACDLS does not report any false alarm.

Chapter 8

Conclusions and Future Work

In this dissertation, we have established the following thesis.

Precise abstract interpretation of hardware Register Transfer Level (RTL) designs can be achieved by CDCL-style relational safety analysis of a software representation of RTL.

In doing this, we have made both theoretical and practical contributions to the state-of-the-art in bounded formal verification of hardware RTL designs.

The theoretical contribution include the framework of *Abstract Conflict Driven Learning for Safety* together with the soundness, completeness and termination arguments. ACDLS is an abstract interpretation framework for generalizing CDCL to safety verification over the relational as well as non-relational abstract domains. This generalization has significant practical relevance, namely it enables developing an abstract interpretation tool that can *efficiently* and *precisely* reason about disjunctive properties over the non-distributive abstract domains. The precision comes from the automatic transformer refinement using learning and the efficiency is due to the use of the non-distributive abstract domains.

The practical contributions of this dissertation are multi-fold. The first contribution presents an automatic translation of hardware designs in Verilog RTL into software netlist in C for the purposes of formal verification in Chapter 3. The second contribution explores the application of native software analyzers including abstract interpretation for bounded as well as unbounded property verification of the software netlist design generated from the RTL in Chapter 4. Chapter 4 summarizes that abstract interpretation with manual guidance is effective for property verification of the software netlist designs. The third contribution presents a comparison of monolithic and path-based symbolic execution techniques for bounded safety verification of the software netlist designs in Chapter 5. Chapter 5 summarizes that path-based symbolic execution performs better than monolithic SAT/SMT-based BMC for proving bounded safety as well as detecting bugs in a floating-point arithmetic circuits, an

UART design and a System-on-a-chip design. The fourth contribution instantiates ACDLS over the template polyhedra abstract domain for precise and efficient bounded safety verification in Chapter 7. Chapter 7 summarizes that the analysis using ACDLS is faster than a bit-level model checker, CBMC, and sufficiently more precise than a commercial abstract interpreter, Astrée.

Our novel verification framework enables technology transfer from the program analysis research as well as satisfiability research to hardware verification. It is apparent that the verification techniques presented in this dissertation open up further avenues for technology transfer between program analysis research and hardware verification research. For example, ACDLS can also be used to verify hardware designs at higher level of abstractions, such as microarchitectural level or behavioral level, as well as systems software or device drivers.

Future Work

In this chapter, we identify few avenues for future research.

Precise deduction in ACDLS

Deduction computes a fixed point over an abstract domain, similar to the way in which BCP in propositional solver computes fixed point over the partial assignments domain. In Chapter 7, we show that the precision of the deduction in ACDLS depends on the type of the transformer used – forward, backward or multi-way. ACDLS also computes the best abstract transformer using symbolic formulas for bit-precise reasoning of complex bit-manipulating operations. Furthermore, to make the deduction efficient, ACDLS use lazy closure computation in relational domains. However, new techniques for efficient and precise deduction in ACDLS may impact the performance of the analysis.

Decision Heuristic in ACDLS

Activity-based decision heuristics [167] is a crucial reason for the improved performance of SAT solvers. It may be worth exploring whether extending ACDLS with the similar heuristic gives similar performance benefit. Also, decision heuristics that can exploit the high-level structure of the design, similar to the *ordered* decision heuristic in Chapter 7, may be useful in practice. Using inexpensive static analysis upfront to identify "interesting" branching variables may lead to effective decision making during the model search phase.

Learning techniques in ACDLS

In this dissertation, we use a generalization of the unit rule transformer for learning an abstract transformer over the template polyhedra abstract domain. Compared to learning in propositional solvers, numeric abstract domains such as Intervals, Octagons provide more opportunities for generalizing a conflict reason. An important requirement for learning with generalized unit rule in the non-distributive abstract domains is that the underlying domain must admit precisely complementable meet irreducibles. Furthermore, learning in ACDLS gives an effect of implicit trace partitioning. One of the challenging problems is the identification of alternative learning techniques over arbitrary abstract domains which preserves the property of the learned transformer.

Comparison to other SAT solvers

This dissertation uses MiniSAT solver in the backend of the hardware and software verification tools. Hence, the analysis using ACDLS is compared against the MiniSAT solver only. However, SAT solvers such as Lingeling, zChaff, Glucose and few others implement different decision, propagation, learning and restart strategies. The comparison of ACDLS against these SAT solvers might provide better idea about the impact of different heuristics on the high-level reasoning and the bit-blasted approach.

Domain Refinement in ACDLS

In this dissertation, ACDLS is instantiated with the template polyhedra abstract domain which supports a product of the Boolean domain and numerical abstract domain such as Intervals, Octagons and fixed co-efficient polyhedra. However, the domain in ACDLS is fixed a-priori, and it never changes during the analysis. An alternative option would be to start the analysis with an inexpensive domain and only switch to a more expensive domain whenever the current domain is insufficient to prove the property.

Unbounded verification

Chapter 7 presents an instantiation of ACDLS for bounded property verification of RTL designs. However, unbounded verification using ACDLS can be achieved by leveraging ACDLS with effective invariant generation techniques. This entails handling loops explicitly (without unrolling) through a controlled gradual widening approach that delays the widening to some later point in the analysis or employs widening with certain thresholds.

Bibliography

[1] http://minisat.se/.

[2] http://fmv.jku.at/lingeling/.

[3] http://fmv.jku.at/boolector/.

[4] IEEE 1364-2001 verilog hardware description language. http://ieeexplore.ieee.org/xpl/aboutJournal.jsp?punumber=7578/.

[5] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the Symposium on Principles of Programming Languages, POPL*, pages 1–11. ACM, 1988.

[6] Nina Amla, Xiaoqun Du, Andreas Kuehlmann, Robert P. Kurshan, and Kenneth L. McMillan. An analysis of sat based model checking techniques in an industrial environment. In *Proceedings of the Correct Hardware Design and Verification Methods, CHARME*, pages 254–268. Springer, 2005.

[7] Zaher S. Andraus, Mark H. Liffiton, and Karem A. Sakallah. Reveal: A formal verification tool for Verilog designs. In *Proceedings of the Logic for Programming, Artificial Intelligence, and Reasoning, LPAR*, volume 5330, pages 343–352. Springer, 2008.

[8] Bahareh Badban, Jaco van de Pol, Olga Tveretina, and Hans Zantema. Generalizing DPLL and satisfiability for equalities. *Information and Computation*, 205(8):1188–1211, 2007.

[9] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[10] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the Programming Language Design and Implementation, PLDI*, pages 203–213. ACM, 2001.

[11] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *Proceedings of the Symposium on Principles of Programming Languages, POPL*, pages 1–3. ACM, 2002.

[12] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proceedings of the Computer Aided Verification, CAV*, pages 171–177. Springer, 2011.

[13] Clark Barrett, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Splitting on demand in SAT modulo theories. In *Proceedings of the Logic for Programming, Artificial Intelligence, and Reasoning, LPAR*, pages 512–526, 2006.

[14] Karen A. Bartlett, Robert K. Brayton, Gary D. Hachtel, Reily M. Jacoby, Christopher R. Morrison, Richard L. Rudell, Alberto L. Sangiovanni-Vincentelli, and Albert R. Wang. Multi-level logic minimization using implicit don't cares. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 7(6):723–740, 1988.

[15] Jason Baumgartner and Andreas Kuehlmann. Enhanced diameter bounding via structural. In *Proceedings of the Conference on Design, Automation and Test in Europe, DATE*, pages 36–41, 2004.

[16] Jason Baumgartner, Hari Mony, Viresh Paruthi, Robert Kanzelman, and Geert Janssen. Scalable sequential equivalence checking across arbitrary design transformations. In *Proceedings of the International Conference on Computer Design, ICCD*, pages 259–266. IEEE, 2006.

[17] Julien Bertrane, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Static analysis by abstract interpretation of embedded critical software. *ACM SIGSOFT Software Engineering Notes*, 36(1):1–8, 2011.

[18] Julien Bertrane, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Static analysis and verification of aerospace software by abstract interpretation. *Foundations and Trends in Programming Languages*, 2(2-3):71–190, 2015.

[19] Julien Bertrane, Patrick Cousot, Radhia Cousot, Jrme Feret, Laurent Mauborgne, Antoine Min, and Xavier Rival. Static analysis and verification of aerospace software by abstract interpretation. In *AIAA Infotech@Aerospace*, 2010.

[20] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A tool for configurable software verification. In *Proceedings of the Computer Aided Verification, CAV*, pages 184–190. Springer, 2011.

[21] A. Biere, M. Heule, H. Van Maaren, and T. Walsh. *Handbook of Satisfiability*. IOS press, 2009.

[22] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.

[23] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, pages 193–207. Springer, 1999.

[24] Per Bjesse. A practical approach to word level model checking of industrial netlists. In *Proceedings of the Computer Aided Verification, CAV*, pages 446–458. Springer, 2008.

[25] Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In *Fields of Logic and Computation II, FLC*, pages 24–51. Springer, 2015.

[26] Nikolaj Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. On solving universally quantified horn clauses. In *Proceedings of the Static Analysis Symposium, SAS*, pages 105–125. Springer, 2013.

[27] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The Essence of Computation*, pages 85–108. Springer, 2002.

[28] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Proceedings of the Programming Language Design and Implementation (PLDI)*, pages 196–207. ACM, 2003.

[29] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. *CoRR*, abs/cs/0701193, 2007.

[30] Aaron R. Bradley. k-step relative inductive generalization. *CoRR*, abs/1003.3649, 2010.

[31] Aaron R. Bradley and Zohar Manna. Checking safety by inductive generalization of counterexamples to induction. In *Proceedings of the Formal Methods in Computer-Aided Design, FMCAD*, pages 173–180. IEEE, 2007.

[32] Martin Brain, Vijay D'Silva, Alberto Griggio, Leopold Haller, and Daniel Kroening. Deciding floating-point logic with abstract conflict driven clause learning. *Formal Methods in System Design*, 45(2):213–245, 2014.

[33] Martin Brain, Vijay D'Silva, Leopold Haller, Alberto Griggio, and Daniel Kroening. An abstract interpretation of DPLL(T). In *Proceedings of the Verification, Model Checking, and Abstract Interpretation, VMCAI*, pages 455–475. Springer, 2013.

[34] Martin Brain, Saurabh Joshi, Daniel Kroening, and Peter Schrammel. Safety Verification and Refutation by k-Invariants and k-Induction. In *Proceedings of the Static Analysis Symposium, SAS*, pages 145–161. Springer, 2015.

[35] Robert K. Brayton, Gary D. Hachtel, A.Vincentelli, F. Somenzi, A. Aziz, S. Cheng, S. A. Edwards, S. P. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R.K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa. VIS: A system for verification and synthesis. In *Proceedings of the Computer Aided Verification, CAV*, pages 428–432. Springer, 1996.

[36] Robert K. Brayton and C McMullen. The decomposition and factorization of boolean expressions. In *Proceedings of the International Symposium on Circuits and Systems, ISCAS*, pages 49–54. IEEE, 1982.

[37] Robert K. Brayton and Alan Mishchenko. ABC: an academic industrial-strength verification tool. In *Proceedings of the Computer Aided Verification, CAV*, pages 24–40. Springer, 2010.

[38] Robert K. Brayton and Alan Mishchenko. ABC: an academic industrial-strength verification tool. In *Proceedings of the Computer Aided Verification, CAV*, pages 24–40. Springer, 2010.

[39] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. *SoftwarePractice & Experience*, 28(8):859–881, 1998.

[40] Randal E. Bryant. Symbolic simulation - techniques and applications. In *Proceedings of the Design Automation Conference, DAC*, pages 517–521. IEEE, 1990.

[41] Randal E. Bryant and Carl-Johan H. Seger. Formal verification of digital circuits using symbolic ternary system models. In *Proceedings of the Computer Aided Verification, CAV*, pages 33–43. Springer, 1990.

[42] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, and David L. Dill. Sequential circuit verification using symbolic model checking. In *Proceedings of the Design Automation Conference, DAC*, pages 46–51. ACM/IEEE, 1990.

[43] Gianpiero Cabodi, Sergio Nocco, and Stefano Quer. Mixing forward and backward traversals in guided-prioritized BDD-based verification. In *Computer Aided Verification, 14th International Conference, CAV*, pages 471–484, 2002.

[44] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the Symposium on Operating Systems Design and Implementation, OSDI*, pages 209–224. USENIX, 2008.

[45] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In *Proceedings of the Computer and Communications Security, CCS*, pages 322–335. ACM, 2006.

[46] S. Chakraborty, Z. Khasidashvili, C. H. Seger, R. Gajavelly, T. Haldankar, D. Chhatani, and R. Mistry. Word-level symbolic trajectory evaluation. In *Proceedings of the Computer Aided Verification, CAV*, pages 128–143. Springer, 2015.

[47] Arun Chandra, Li-C. Wang, and Magdy S. Abadir. Practical considerations in formal equivalence checking of PowerPC microprocessors. In *Proceedings of the Great Lakes Symposium on VLSI, GLSVLSI*, pages 362–367. IEEE, 1998.

[48] Satrajit Chatterjee and Michael Kishinevsky. Automatic generation of inductive invariants from high-level microarchitectural models of communication fabrics. *Formal Methods in System Design*, 40(2):147–169, 2012.

[49] Liqian Chen, Antoine Miné, and Patrick Cousot. A sound floating-point polyhedra abstract domain. In *Proceedings of the Asian Symposium on Programming Languages and Systems, APLAS*, pages 3–18, 2008.

[50] Hana Chockler, Daniel Kroening, and Mitra Purandare. Computing mutation coverage in interpolation-based model checking. *IEEE Transaction on CAD of Integrated Circuits and Systems*, 31(5):765–778, 2012.

[51] Václav Chvátal. Edmonds polytopes and weakly hamiltonian graphs. *Mathematical Programming*, 5(1):29–40, 1973.

[52] Alessandro Cimatti and Alberto Griggio. Software model checking via IC3. In *Proceedings of the Computer Aided Verification, CAV*, pages 277–293. Springer, 2012.

[53] Edmund Clarke and Daniel Kroening. Hardware verification using ANSI-C programs as a reference. In *Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC*, pages 308–311. ACM, 2003.

[54] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, pages 168–176. Springer, 2004.

[55] Edmund Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of the Design Automation Conference, DAC*, pages 368–371. ACM, 2003.

[56] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Language and Systems*, 8(2):244–263, 1986.

[57] Edmund M. Clarke, Masahiro Fujita, Sreeranga P. Rajan, Thomas W. Reps, Subash Shankar, and Tim Teitelbaum. Program slicing of hardware description languages. In *Proceedigns of the Correct Hardware Design and Verification Methods, CHARME*, pages 298–312. Springer, 1999.

[58] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proceedings of the Computer Aided Verification, CAV*, pages 154–169. Springer, 2000.

[59] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[60] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[61] Edmund M. Clarke, Kenneth L. McMillan, Sérgio Vale Aguiar Campos, and Vasiliki Hartonas-Garmhausen. Symbolic model checking. In *Proceedings of the Computer Aided Verification, CAV*, pages 419–427. Springer, 1996.

[62] Edmund M. Clarke, Muralidhar Talupur, Helmut Veith, and Dong Wang. Sat based predicate abstraction for hardware verification. In *Theory and Applications of Satisfiability Testing, SAT*, pages 78–92. Springer Berlin Heidelberg, 2004.

[63] Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, 1976.

[64] Fady Copty, Limor Fix, Ranan Fraer, Enrico Giunchiglia, Gila Kamhi, Armando Tacchella, and Moshe Y. Vardi. Benefits of bounded model checking at an industrial setting. In *Proceedings of the Computer Aided Verification, CAV*, pages 436–453. Springer, 2001.

[65] Scott Cotton. Natural domain SMT: A preliminary assessment. In *Proceedings of the Formal Modeling and Analysis of Timed Systems, FORMATS*, pages 77–91. Springer, 2010.

[66] Olivier Coudert and Jean Christophe Madre. A unified framework for the formal verification of sequential circuits. In *Proceedings of the International Conference on Computer-Aided Design, ICCAD*, pages 126–129. ACM/IEEE, 1990.

[67] Patrick Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoretical Computer Science*, 277(1):47–103, 2002.

[68] Patrick Cousot. Proving the absence of run-time errors in safety-critical avionics code. In *Proceedings of the International Conference on Embedded Software, EMSOFT*, pages 7–9. ACM, 2007.

[69] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Symposium on Principles of Programming Languages, POPL*, pages 238–252. ACM, 1977.

[70] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the Symposium on Principles of Programming Languages, POPL*, pages 269–282. ACM, 1979.

[71] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming*, 13(2&3):103–179, 1992.

[72] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[73] Patrick Cousot and Radhia Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proceedings of the Functional Programming Languages and Computer Architecture, FPCA*, pages 170–181. Springer, 1995.

[74] Patrick Cousot and Radhia Cousot. Refining model checking by abstract interpretation. *Automated Software Engineering*, 6(1), 1999.

[75] Patrick Cousot and Radhia Cousot. Basic concepts of abstract interpretation. pages 359–366. Kluwer Academic Publishers, 2004.

[76] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The Astrée analyzer. In *Proceedings of the European Symposium on Programming, ESOP*, pages 21–30. Springer, 2005.

[77] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the Symposium on Principles of Programming Languages, POPL*, pages 25–35. ACM, 1989.

[78] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[79] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[80] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of ACM*, 7(3):201–215, 1960.

[81] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, pages 337–340. Springer, 2008.

[82] Leonardo Mendonça de Moura and Dejan Jovanovic. A model-constructing satisfiability calculus. In *Proceedings of the Verification, Model Checking, and Abstract Interpretation, VMCAI*, pages 1–12. Springer, 2013.

[83] Leonardo Mendonça de Moura and Dejan Jovanovic. A model-constructing satisfiability calculus. In *Proceedings of the Verification, Model Checking, and Abstract Interpretation, VMCAI*, pages 1–12. Springer, 2013.

[84] Alastair F. Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. Software verification using k-induction. In *Proceedings of the Static Analysis Symposium, SAS*, pages 351–368. Springer, 2011.

[85] Alastair F. Donaldson, Daniel Kroening, and Philipp Rümmer. Automatic analysis of scratch-pad memory code for heterogeneous multicore processors. In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, pages 280–295. Springer, 2010.

[86] Vijay D'Silva, Leopold Haller, and Daniel Kroening. Abstract conflict driven learning. In *Proceedings of the Symposium on Principles of Programming Languages, POPL*, pages 143–154. ACM, 2013.

[87] Vijay D'Silva, Leopold Haller, and Daniel Kroening. Abstract satisfaction. In *Proceedings of the Symposium on Principles of Programming Languages, POPL*, pages 139–150. ACM, 2014.

[88] Vijay D'Silva, Leopold Haller, Daniel Kroening, and Michael Tautschnig. Numeric bounds analysis with conflict-driven learning. In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, pages 48–63. Springer, 2012.

[89] Vijay D'Silva, Daniel Kroening, and Leopold Haller. Satisfiability solvers are static analysers. In *Proceedings of the Static Analysis Symposium, SAS*, pages 317–333. Springer, 2012.

[90] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *Proceedings of the Theory and Applications of Satisfiability Testing, SAT*, pages 61–75. Springer, 2005.

[91] Cindy Eisner and Dana Fisman. *A Practical Introduction to PSL.* Springer, 2006.

[92] Javier Esparza and Stefan Schwoon. A BDD-based model checker for recursive programs. In *Proceedings of the Computer Aided Verification, CAV*, volume 1, pages 324–336. Springer, 2001.

[93] P.M. Farmwald. *On the design of high performance digital arithmetic units*. PhD thesis, Stanford University, 1981.

[94] John Howard Eli Fiskio-Lasseter and Amr Sabry. Putting operational techniques to the test: A syntactic theory for behavioral verilog. *Electronic Notes on Theoretical Computer Science*, 26:34–51, 1999.

[95] D. Forte, Swarup Bhunia, and M. M. Tehranipoor. *Hardware Protection through Obfuscation*. Springer, 2017.

[96] Uri Frank, Tsachy Kapschitz, and Ran Ginosar. A predictive synchronizer for periodic clock domains. *Formal Methods in System Design*, 28(2):171–186, 2006.

[97] Martin Frnzle, Christian Herde, Tino Teige, Stefan Ratschan, and Tobias Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *Journal on Satisfiability, Boolean Modeling and Computation*, 1:209–236, 2007.

[98] Masahiro Fujita. Verification of arithmetic circuits by comparing two similar circuits. In *Proceedings of the Computer Aided Verification, CAV*, volume 1102, pages 159–168. Springer, 1996.

[99] Malay K. Ganai, Praveen Yalagandula, Adnan Aziz, Andreas Kuehlmann, and Vigyan Singhal. SIVA: A system for coverage-directed state space search. *Journal of Electronic Testing*, 17(1):11–27, 2001.

[100] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): fast decision procedures. In *Proceedings of the Computer Aided Verification, CAV*, pages 175–188. Springer, 2004.

[101] Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In *Proceedings of the Computer Aided Verification, CAV*, pages 53–65. Springer, 2001.

[102] Thomas Gawlitza and Helmut Seidl. Precise fixpoint computation through strategy iteration. In *Proceedings of the Programming Languages and Systems Symposium*, pages 300–315. Springer Berlin Heidelberg, 2007.

[103] Thomas M. Gawlitza and Helmut Seidl. Precise relational invariants through strategy iteration. In *Proceedings of the Computer Science Logic Workshop, CSL*, volume 4646, pages 23–40. Springer, 2007.

[104] Michael P. Gerlek, Eric Stoltz, and Michael Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven ssa form. *ACM Transactions on Programming Languages and Systems*, 17(1):85–122, 1995.

[105] Roberto Giacobazzi and Elisa Quintarelli. Incompleteness, counterexamples, and refinements in abstract model-checking. In *SAS*, pages 356–373. Springer, 2001.

[106] Roberto Giacobazzi and Francesco Ranzato. Refining and compressing abstract domains. *Automata, Languages and Programming*, pages 771–781, 1997.

[107] Roberto Giacobazzi and Francesco Ranzato. Optimal domains for disjunctive abstract interpretation. *Science of Computer Programming*, 32(1-3):177–210, 1998.

[108] Tilman Glökler, Jason Baumgartner, Devi Shanmugam, A. E. (Rick) Seigler, Gary A. Van Huben, Barinjato Ramanandray, Hari Mony, and Paul Roessler. Enabling large-scale pervasive logic verification through multi-algorithmic formal reasoning. In *Proceedings of the Formal Methods in Computer-Aided Design, FMCAD*, pages 3–10. IEEE, 2006.

[109] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the Programming Language Design and Implementation, PLDI*, pages 213–223. ACM, 2005.

[110] Ralph E. Gomory. *Outline of an Algorithm for Integer Solutions to Linear Programs and An Algorithm for the Mixed Integer Problem*, pages 77–103. Springer Berlin Heidelberg, 2010.

[111] Denis Gopan and Thomas Reps. Guided static analysis. In *Proceedings of the Static Analysis Symposium, SAS*, pages 349–365. Springer, 2007.

[112] Mike Gordon, Thomas Kropf, and Dirk Hoffman. Semantics of the intermediate language il. `http://www.cl.cam.ac.uk/~mjcg/IL/IL15.ps`, 1999. Technical Report D2.1c, PROSPER.

[113] Shankar G. Govindaraju and David L. Dill. Verification by approximate forward and backward reachability. In *IEEE/ACM International Conference on Computer-Aided Design, ICCAD*, pages 366–370, 1998.

[114] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *Proceedings of the Computer Aided Verification, CAV*, pages 72–83. Springer, 1997.

[115] George Gratzer. *Lattice Theory: Foundation*. Birkhauser Basel, 2011.

[116] David J. Greaves. A verilog to C compiler. In *Proceedings of the International Workshop on Rapid System Prototyping, RSP*, pages 122–127. IEEE, 2000.

[117] Jan Friso Groote and Tim A. C. Willemse. Parameterised boolean equation systems. *Theoretical Computer Science*, 343(3):332–369, 2005.

[118] Bhargav S. Gulavani, Supratik Chakraborty, Aditya V. Nori, and Sriram K. Rajamani. Automatically refining abstract interpretations. In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, pages 443–458. Springer, 2008.

[119] Bhargav S. Gulavani and Sriram K. Rajamani. Counterexample driven refinement for abstract interpretation. In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, pages 474–488. Springer, 2006.

[120] Sumit Gulwani and Ashish Tiwari. Combining abstract interpreters. In *Proceedings of the Programming Language Design and Implementation, PLDI*, pages 376–386. ACM, 2006.

[121] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The SeaHorn verification framework. In *Proceedings of the Computer Aided Verification, CAV*, volume 9206, pages 343–361. Springer, 2015.

[122] Leopold Carl Robert Haller. *Abstract satisfaction*. PhD thesis, University of Oxford, UK, 2013.

[123] Maria Handjieva and Stanislav Tzolovski. Refining static analyses by trace-based partitioning using control flow. In *Proceedings of the Static Analysis Symposium, SAS*, pages 200–214. Springer, 1998.

[124] William R. Harris, Sriram Sankaranarayanan, Franjo Ivančić, and Aarti Gupta. Program analysis via satisfiability modulo path programs. In *Proceedings of the Symposium on Principles of Programming Languages, POPL*, pages 71–82. ACM, 2010.

[125] Matthias Heizmann, Daniel Dietsch, Marius Greitschus, Jan Leike, Betim Musa, Claus Schätzle, and Andreas Podelski. Ultimate automizer with two-track proofs - (competition contribution). In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, pages 950–953. Springer, 2016.

[126] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Software model checking for people who love automata. In *Proceedings of the Computer Aided Verification, CAV*, pages 36–52. Springer, 2013.

[127] S. Hendricx and L. Claesen. Formally verified redundancy removal. In *Proceedings of the Design, Automation and Test in Europe, DATE*, pages 150–155. IEEE/ACM, 1999.

[128] Gerard Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, first edition, 2003.

[129] Himanshu Jain, Daniel Kroening, Natasha Sharygina, and Edmund M. Clarke. Word level predicate abstraction and refinement for verifying RTL verilog. In *Proceedings of the Design Automation Conference, DAC*, pages 445–450. ACM, 2005.

[130] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *Proceedings of the Computer Aided Verification, CAV*, pages 661–667. Springer, 2009.

[131] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *Proceedings of the Computer Aided Verification, CAV*, pages 661–667. Springer, 2009.

[132] R.B. Jones. *Applications of Symbolic Simulation to the Formal Verification of Microprocessors*. PhD thesis, Stanford University, 1999.

[133] D. Jovanovic, C. Barrett, and L. de Moura. The design and implementation of the model constructing satisfiability calculus. In *Proceedings of the Formal Methods in Computer-Aided Design (FMCAD)*, pages 173–180. IEEE, 2013.

[134] Dejan Jovanović and Leonardo de Moura. Cutting to the chase solving linear integer arithmetic. In *Proceedings of the International Conference on Automated Deduction, CADE*, pages 338–353. Springer, 2011.

[135] Dejan Jovanovic and Leonardo de Moura. Solving non-linear arithmetic. *ACM Communications in Computer Algebra*, 46(3/4):104–105, 2012.

[136] Roberto J. Bayardo Jr. and Robert Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Association for the Advancement of Artificial Intelligence, AAAI*, pages 203–208. AAAI Press/ The MIT Press, 1997.

[137] Hans W. Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, UCLA, 1968.

[138] M. Keating. *The Simple Art of SoC Design*. Springer, 2011.

[139] Michael Keating, David Flynn, Robert Aitken, Alan Gibbons, and Kaijian Shi. *Low Power Methodology Manual*. Springer US, 2007.

[140] A. Koelbl, J. H. Kukula, and R. Damiano. Symbolic RTL simulation. In *Proceedings of the Design Automation Conference, DAC*, pages 47–50. ACM, 2001.

[141] Alfred Kölbl, Reily Jacoby, Himanshu Jain, and Carl Pixley. Solver technology for system-level to RTL equivalence checking. In *Proceedings of the Design, Automation and Test in Europe, DATE*, pages 196–201. IEEE, 2009.

[142] Daniel Kroening, Edmund Clarke, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of the Design Automation Conference, DAC*, pages 368–371. ACM, 2003.

[143] Daniel Kroening and Sanjit A. Seshia. Formal verification at higher levels of abstraction. In *Proceedings of the International Conference on Computer-Aided Design, ICCAD*, pages 572–578. IEEE, 2007.

[144] Daniel Kroening and Ofer Strichman. *Decision Procedures*. Springer, 2008.

[145] Daniel Kroening and Georg Weissenbacher. Lifting propositional interpolants to the word-level. In *Proceedings of the Formal Methods in Computer-Aided Design, FMCAD*, pages 85–89, 2007.

[146] Daniel Kroening and Georg Weissenbacher. Interpolation-based software verification with wolverine. In *Proceedings of the Computer Aided Verification, CAV*, pages 573–578. Springer, 2011.

[147] Sava Krstic and Amit Goel. Architecting solvers for SAT modulo theories: Nelson-oppen with DPLL. In *Proceedings of the International Symposium of Frontiers of Combining Systems, FroCoS*, pages 1–27. Springer, 2007.

[148] Andreas Kuehlmann and Cornelis AJ van Eijk. *Combinational and sequential equivalence checking*. Springer, 2002.

[149] Orna Kupferman and Moshe Y. Vardi. An automata-theoretic approach to reasoning about infinite-state systems. In *Proceedings of the Computer Aided Verification, CAV*, pages 36–52. Springer, 2000.

[150] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. In *Proceedings of the Programming Language Design and Implementation, PLDI*, pages 193–204. ACM, 2012.

[151] Akash Lal and Shaz Qadeer. Reachability modulo theories. In *Proceedings of the Reachability Problems Workshop, RP*, pages 23–44. Springer, 2013.

[152] Juncao Li, Fei Xie, Thomas Ball, and Vladimir Levin. Efficient reachability analysis of Büchi pushdown systems for hardware/software co-verification. In *Proceedings of the Computer Aided Verification, CAV*, pages 339–353. Springer, 2010.

[153] Lingyi Liu and Shobha Vasudevan. Scaling input stimulus generation through hybrid static and dynamic analysis of RTL. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 20(1):4:1–4:33, 2014.

[154] A. Mader. *Verification of Modal Properties Using Boolean Equation Systems*. PhD thesis, Technical University of Munich, 1997.

[155] Angelika Mader. Modal μ-calculus, model checking and gauß elimination. In *Tools and Algorithms for Construction and Analysis of Systems, TACAS*, pages 72–88, 1995.

[156] Vasco M. Manquinho and João P. Marques Silva. Effective lower bounding techniques for pseudo-boolean optimization. In *Proceedings of the Design, Automation and Test in Europe DATE*, pages 660–665, 2005.

[157] J.P. Marques-Silva. *Search Algorithms for Satisfiability Problems in Combinational Switching Circuits*. PhD thesis, University of Michigan, USA, 1995.

[158] Kenneth L. McMillan. Interpolation and SAT based model checking. In *Proceedings of the Computer Aided Verification, CAV*, pages 1–13. Springer, 2003.

[159] Kenneth L. McMillan. Lazy abstraction with interpolants. In *Proceedings of the Computer Aided Verification, CAV*, pages 123–136. Springer, 2006.

[160] Kenneth L. Mcmillan, Andreas Kuehlmann, and Mooly Sagiv. Generalizing DPLL to richer logics. In *Proceedings of the Computer Aided Verification, CAV*, pages 462–476. Springer, 2009.

[161] Joke Meheus and Diderik Batens. A formal logic for abductive reasoning. *Logic Journal of the IGPL*, 14(2):221–236, 2006.

[162] Baoluo Meng, Andrew Reynolds, Cesare Tinelli, and Clark W. Barrett. Relational constraint solving in SMT. In *Proceedings of the International Conference on Automated Deduction, CADE*, pages 148–165. Springer, 2017.

[163] Antoine Miné. *Weakly Relational Numerical Abstract Domains. (Domaines numériques abstraits faiblement relationnels)*. PhD thesis, École Polytechnique, Palaiseau, France, 2004.

[164] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[165] Antoine Miné, Jason Breck, and Thomas W. Reps. An algorithm inspired by constraint solvers to infer inductive invariants in numeric programs. In *Proceedings of the European Symposium on Programming, ESOP*, volume 9632 of *LNCS*, pages 560–588. Springer, 2016.

[166] David Monniaux and Laure Gonnord. Using bounded model checking to focus fixpoint iterations. In *Proceedings of the Static Analysis Symposium, SAS*, pages 369–385. Springer, 2011.

[167] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference, DAC*, pages 530–535. ACM, 2001.

[168] Steven S. Muchnick and Neil D. Jones. *Program Flow Analysis: Theory and Application*. Prentice Hall Professional Technical Reference, 1981.

[169] Rajdeep Mukherjee, Pallab Dasgupta, Ajit Pal, and Subhankar Mukherjee. Formal verification of hardware/software power management strategies. In *Proceedings of the International Conference on VLSI Design, VLSID*, pages 326–331. IEEE, 2013.

[170] Rajdeep Mukherjee, Saurabh Joshi, Andreas Griesmayer, Daniel Kroening, and Tom Melham. Equivalence checking of a floating-point unit against a high-level C model.

In *Proceedings of the International Symposium on Formal Methods, FM*, pages 551–558. Springer, 2016.

[171] Rajdeep Mukherjee, Daniel Kroening, and Tom Melham. Hardware verification using software analyzers. In *Proceedings of the Annual Symposium on VLSI, ISVLSI*, pages 7–12. IEEE, 2015.

[172] Rajdeep Mukherjee, Daniel Kroening, Tom Melham, and Mandayam Srivas. Equivalence checking using trace partitioning. In *Proceedings of the Annual Symposium on VLSI, ISVLSI*, pages 13–18. IEEE, 2015.

[173] Rajdeep Mukherjee, Mitra Purandare, Raphael Polig, and Daniel Kroening. Formal techniques for effective co-verification of hardware/software co-designs. In *Proceedings of the Design Automation Conference, DAC*, pages 35:1–35:6. ACM, 2017.

[174] Rajdeep Mukherjee, Peter Schrammel, Leopold Haller, Daniel Kroening, and Tom Melham. Lifting CDCL to template-based domains for program verification. In *Proceedings of the Automated Technology for Verification and Analysis, ATVA*. Springer, 2017.

[175] Rajdeep Mukherjee, Michael Tautschnig, and Daniel Kroening. v2c – A verilog to C translator. In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, pages 580–586. Springer, 2016.

[176] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages Systems*, 1(2):245–257, 1979.

[177] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.

[178] Gordon J. Pace. The semantics of verilog using transition system combinators. In *Proceedings of the Formal Methods in Computer-Aided Design, FMCAD*, pages 405–422. Springer, 2000.

[179] Marie Pelleau, Antoine Miné, Charlotte Truchet, and Frédéric Benhamou. A constraint solver based on abstract domains. In *Proceedings of the Verification, Model Checking, and Abstract Interpretation, VMCAI*, volume 7737, pages 434–454. Springer, 2013.

[180] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Proceedings of the Tools and Algorithms for Construction and Analysis of Systems, TACAS*, pages 151–166. Springer, 1998.

[181] Thomas W. Reps, Shmuel Sagiv, and Greta Yorsh. Symbolic implementation of the best transformer. In *Proceedings of the Verification, Model Checking, and Abstract Interpretation, VMCAI*, pages 252–266. Springer, 2004.

[182] Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM Transactions on Programming Languages and Systems, TOPLAS*, 29(5), 2007.

[183] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[184] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the Symposium on Principles of Programming Languages, POPL*, pages 12–27. ACM, 1988.

[185] Sriram Sankaranarayanan, Michael Colón, Henny B. Sipma, and Zohar Manna. Efficient strongly relational polyhedral analysis. In *Proceedings of the Verification, Model Checking, and Abstract Interpretation, VMCAI*, pages 111–125. Springer, 2006.

[186] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Scalable analysis of linear systems using mathematical programming. In *Proceedings of the Verification, Model Checking, and Abstract Interpretation, VMCAI*, pages 25–41. Springer, 2005.

[187] J Santos and VM Manquinho. Learning techniques for pseudo-boolean solving. In *In the Workshop of the Logic for Programming, Artificial Intelligence, and Reasoning, LPAR*, 2008.

[188] Hisashi Sasaki. A formal semantics for verilog-vhdl simulation interoperability by abstact state machine. In *Proceedings of the Design, Automation and Test in Europe, DATE*, page 353. IEEE/ACM, 1999.

[189] David A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *Proceedings of Symposium on Principles of Programming Languages, POPL*, pages 38–48. ACM, 1998.

[190] Peter Schrammel and Daniel Kroening. 2LS for program analysis. In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, pages 905–907. Springer, 2016.

[191] B. Selman, H. Kautz, and B. Cohen. Local search strategies for satisfiability testing. *DIMACS series in Discrete Mathematics and Theoretical Computer Science*, 26, 1996.

[192] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT solver. In *Proceedings of the Formal Methods in Computer-Aided Design, FMCAD*, pages 108–125. Springer, 2000.

[193] Mary Sheeran and Gunnar Stålmarck. A tutorial on Stålmarcks's proof procedure for propositional logic. In *Proceedings of the Formal Methods in Computer-Aided Design, FMCAD*, pages 82–99. Springer, 1998.

[194] Hossein M. Sheini and Karem A. Sakallah. Pueblo: A modern pseudo-boolean SAT solver. In *Proceedings of the Design, Automation and Test in Europe, DATE*, pages 684–685, 2005.

[195] João P. Marques Silva and Karem A. Sakallah. Dynamic search-space pruning techniques in path sensitization. In *Proceedings of the Design Automation Conference, DAC*, pages 705–711. ACM, 1994.

[196] João P. Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[197] Gagandeep Singh, Markus Püschel, and Martin T. Vechev. Making numerical program analysis fast. In *Proceedings of the Programming Language Design and Implementation, PLDI*, pages 303–313. ACM, 2015.

[198] A. Prasad Sistla and Edmund M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, 1985.

[199] Jiri Slaby, Jan Strejcek, and Marek Trtík. Compact symbolic execution. In *Proceedings of the Automated Technologies for Verification and Analysis, ATVA*, pages 193–207. Springer, 2013.

[200] Jiri Slaby, Jan Strejcek, and Marek Trtík. Symbiotic: Synergy of instrumentation, slicing, and symbolic execution - (competition contribution). In *Proceedings of the*

Tools and Algorithms for the Construction and Analysis of Systems, TACAS, pages 630–632. Springer, 2013.

[201] Fabio Somenzi and Roderick Bloem. Efficient büchi automata from LTL formulae. In *Proceedings of the Computer Aided Verification, CAV*, pages 248–263. Springer, 2000.

[202] Niklas Sörensson and Armin Biere. Minimizing learned clauses. In *Proceedings of the Theory and Applications of Satisfiability Testing, SAT*, pages 237–243. Springer, 2009.

[203] Y. N. Srikant and Priti Shankar. *The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition*. CRC Press, Inc., 2nd edition, 2007.

[204] Dominik Stoffel and Wolfgang Kunz. Equivalence checking of arithmetic circuits on the arithmetic bit level. *IEEE Transactions on CAD of Integrated Circuits and Systems, TCAD*, 23(5):586–597, 2004.

[205] Pramod Subramanyan, Yakir Vizel, Sayak Ray, and Sharad Malik. Template-based synthesis of instruction-level abstractions for soc verification. In *Proceedings of the Formal Methods in Computer-Aided Design, FMCAD*, pages 160–167. IEEE, 2015.

[206] A. Sülflow, U. Kühne, G. Fey, D. Große, and Rolf Drechsler. Wolfram- A word level framework for formal verification. In *Proceedings of the International Symposium on Rapid System Prototyping, RSP*, pages 11–17. IEEE, 2009.

[207] Sasidhar Sunkari, Supratik Chakraborty, Vivekananda M. Vedula, and Kailasnath Maneparambil. A scalable symbolic simulator for Verilog RTL. In *Proceedings of the Microprocessor Test and Verification Workshop, MTV*, pages 51–59. IEEE, 2007.

[208] Vaibbhav Taraate. *Digital Logic Design Using Verilog*. Springer India, 2016.

[209] Aditya V. Thakur and Thomas W. Reps. A method for symbolic computation of abstract operations. In *Proceedings of the Computer Aided Verification - 24th International Conference, CAV*, pages 174–192. Springer, 2012.

[210] Ashish Tiwari and Sumit Gulwani. Logical interpretation: Static program analysis using theorem proving. In *Proceedings of the International Conference on Automated Deduction, CADE*, pages 147–166. Springer, 2007.

[211] Rachel Tzoref and Orna Grumberg. Automatic refinement and vacuity detection for symbolic trajectory evaluation. In *Proceedings of the Computer Aided Verification, CAV*, pages 190–204, 2006.

[212] C. A. J. van Eijk. Sequential equivalence checking without state space traversal. In *Proceedings of the Design, Automation and Test in Europe, DATE*, pages 618–623. IEEE, 1998.

[213] Moshe Y. Vardi. *Alternating Automata and Program Verification*. Springer-Verlag, 1995.

[214] S. Vasudevan. *High Level Static Analysis of System Descriptions for Taming Verification Complexity*. PhD thesis, The University of Texas at Austin, 2007.

[215] Bart Vergauwen and Johan Lewi. Efficient local correctness checking for single and alternating boolean equation systems. In *International Colloquium on Automata, Languages and Programming, ICALP*, pages 304–315, 1994.

[216] Björn Wachter, Daniel Kroening, and Joël Ouaknine. Verifying multi-threaded software with impact. In *Proceedings of the Formal Methods in Computer-Aided Design, FMCAD*, pages 210–217. IEEE, 2013.

[217] Mark Weiser. Program slicing. In *Proceedings of the International Conference on Software Engineering, ICSE*, pages 439–449. IEEE, 1981.

[218] Tobias Welp and Andreas Kuehlmann. Property directed invariant refinement for program verification. In *Proceedings of the Design, Automation & Test In Europe, DATE*, pages 1–6. IEEE, 2014.

[219] Chris Wilson and David L. Dill. Reliable verification using symbolic simulation with scalar values. In *Proceedings of the Design Automation Conference, DAC*, pages 124–129. ACM, 2000.

[220] Bin Xue, Prosenjit Chatterjee, and Sandeep K. Shukla. Simplification of C-RTL equivalent checking for fused multiply add unit using intermediate models. In *Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC*, pages 723–728. IEEE, 2013.

[221] Bwolen Yang, Randal Bryant, David OHallaron, Armin Biere, Olivier Coudert, Geert Janssen, Rajeev Ranjan, and Fabio Somenzi. A performance study of BDD-based

model checking. In *Proceedings of the Formal Methods in Computer-Aided Design, FMCAD*, pages 533–533. Springer, 1998.

[222] C. Han Yang and David L. Dill. Validation with guided search of the state space. In *Proceedings of the Design Automation Conference, DAC*, pages 599–604, 1998.

[223] Ramin Zabih and David A. McAllester. A rearrangement search strategy for determining propositional satisfiability. In *Proceedings of the National Conference on Artificial Intelligence*, pages 155–160. AAAI Press/ The MIT Press, 1988.

[224] Hantao Zhang. SATO: an efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction, CADE*, pages 272–275. Springer, 1997.

[225] L Zhang. *Searching the Truth: Techniques for Satisfiability of Boolean formulas*. PhD thesis, Princeton University, USA, 2003.

[226] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In *Proceedings of the International Conference on Computer-Aided Design, ICCAD*, pages 279–285. IEEE, 2001.