The Dissertation Committee for Fadi A. Zaraket

certifies that this is the approved version of the following dissertation:

Program Analysis with Boolean Logic Solvers

Committee:

Adnan Aziz, Supervisor

Jacob Abraham

Jason Baumgartner

Sarfraz Khurshid

Vladimir Lifschitz

Program Analysis with Boolean Logic Solvers

by

Fadi A. Zaraket, B.E;M.E.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2007

إلى والدي العزيز عبد المجيد، معلمي الأوّل
إلى والدتي إسعاف، سرّ الحنان
إلى إخوتي مها وأُميمة وعلي، شركائي في الذكريات
إلى ميّاسة الحبيبة، شريكة و بهجة حياتي
إلى ابنتي ياسمين، الآمنة التي تنبض فرحاً
إلى مارغريدا جاكوم، وكلّ من سبقنا وتوقف عن الانتظار

Translation: To my dear father *Abdul Majid*, my best teacher,

To my mother *Isaaf*, the secret of tenderness,

To my siblings *Maha, Oumayma, and Ali*, my memory partners,

To my beloved wife *Mayassah*, the joy of my life,

To my daughter *Yasmin*, always spreading happiness,

To *Margarida Jacome* and all those who stopped waiting, and reached before us.

Acknowledgments

بِسْمِ اللّهِ الرَّحْمـن الرَّحِيمِ، والْحَمْدُ للّهِ رَبِّ الْعَالَمِينَ، والصلاة والسلام على محمَّدٍ وآله الطاهرين.

I start by thanking God for the continuous gifts that allowed me a chance to continue my graduate studies and an opportunity to meet all the nice people who helped me finish this work. The art of acknowledgment, العِرفَان in Arabic, is the science of ethics that appreciates the awareness of one's existence. To practice it is to prove that the acquisition of knowledge is a worthwhile journey. For that, I would like to extend my sincere appreciation to all those who contributed to this work with the hope that they would excuse me if I forgot to name them or did not address them with appropriate grace.

I will always be in debt for my adviser, Adnan Aziz, due to the guidance and support he offered me for the last four years. Adnan's elegant and accurate approach to formalisms guided the development of my logical reasoning. This shortened the learning curve for me and helped me acquire all that I know about verification. His encouragement and the enthusiasm he showed for my initiatives helped me gain the confidence necessary to challenge traditional approaches and pursue my own ideas. I hope my graduation from UT becomes an additional reason for me to keep in contact with Adnan since I can not imagine I will be able to find an equal guidance.

I am full of gratitude to Sarfraz Khurshid who opened the door of software verification for me. Sarfraz gave me suggestions on how to extend the origin of this work to what it is now and helped me with countless discussions and insights.

I would like to thank Jason Baumgartner for his mentorship all along this effort. He was always available for intelligent discussions and always ready with valuable insight across the different phases of this work. I was fortunate to meet and attend to Jacob Abraham's advice when I first joined the University of Texas at Austin. His wisdom was instrumental and helped me focus early on the direction I wanted to take. I owe Vladimir Lifschitz the art of presenting math and mathematical proofs in front of an audience. In his brilliant teaching methods, he tutored and listened to us while we acquired his rigorous mathematical tongue on logic.

Margarida Jacome was an inspiration to all of us for what a sincere scholar and a devoted researcher she was. Despite her terminal cancer, she cared for the future and success of her students to the last minute. Margarida was helpful in every way a professor and a friend can be. I was honored to meet her and be her student and I feel great sadness that she left us early. I hope the dedication of this work to her memory does some justice to her invaluable contribution and I find some comfort in that I was privileged to co-author with her one of her last contributions.

I must offer special thanks to all the good people who make the University of Texas at Austin Computer Engineering graduate program work, especially Mellanie Gullick who spares no effort in helping a student feel at home. UT is definitely the place to be for a graduate student and one is privileged to spend his graduate years amongst its ranks.

Many thanks to my managers at IBM, especially David Schrader, Sue Beck and Sam Robertson who supported me throughout the last four years and approved the funding of my studies through the Degree Work and Study Program at IBM. I would like also to extend many thanks to my friends and colleagues in my IBM team Wolfgang Roesner, Jason Baumgartner, Viresh Paruthi, Geert Janssen, Jessie Xu, Mark Williams, Hari Mony, Robert Kanzelman, Stephen Bergman, John Krauss, Robert Shadowen, Zoltan Hidvegi, Matyäs Sustik, and Ali El-Zein for their contributions to the verification framework used to conduct the experiments of this work and their support of my education goals.

A word of appreciation is due to the researchers who contributed to the related work referenced in the bibliography of this dissertation. Daniel Jackson, Emina Torlak and the Alloy team at MIT were very kind to provide their insight. I thank Edmund Clark, Daniel Kroening, and the CBMC team as well as Rajish Gupta, Sumit Gupta, Nikil Dutt, Alex Nicolau and the SPARK team for releasing their tools.

رَبِّ ارْحَمْهُمَا كَمَا رَبَّيَانِي صَغِيرًا *(My lord see both of them with mercy as they both raised me and cared for me when I was little)* is a verse that serves my gratitude to my father, Abdul-Majid, and my mother, Isaaf. They both deserve most of the credit for not giving up on my return to academic ranks from industry while I often lost hope to act on the delayed dream. It would not have been possible to conceive the restart of my journey towards a Doctorate of Philosophy without their constant encouragement. They instilled the academic tradition in me, my sisters Maha and Oumayma, and my brother Ali, by offering us practical examples on how to hold the exchange of knowledge as the focal value in one's life. Along with my sisters and brother they were indispensable to the success of this work. I also extend my thanks to my in-laws, Basil, Yasmin, Zahra, Hassan, and Farah for their prayers and support.

Very essential to the success of my return to academy ranks were my friends; in particular Ali El-Zein, Hussein Sharafeddine, Rabih Zbib, and Amal Rahmeh. I owe them all for their friendship and help and all the time they spent discussing ideas with me.

Being a part-time PhD student and a full-time working father is an adventure that pushes a man to his creativity and production limits. The fact that this experience was actually smooth and enjoyable was due to the sacrifices, the insistence, and the high morale of my beloved wife Mayassah. She created and kept a warm house where her smiles and tenderness melted all my frustrations and where her enthusiasm and faith in me turned my frequent impatience into motivation and energy. My endless thanks will not be enough for the precious time she made for me while caring for me and for our daughter Yasmin.

FADI A. ZARAKET

The University of Texas at Austin

December 2007

Program Analysis with Boolean Logic Solvers

Publication No. _____

Fadi A. Zaraket, Ph.D.

The University of Texas at Austin, 2007

Supervisor: Adnan Aziz

Modern computation systems are very complex. As a result they are very challenging to reason about and validate. Validation techniques based on dynamic analysis—testing— are time consuming and offer no guarantees of correctness. Researchers have proposed static verification techniques which require no testing and are complete. Existing static verification techniques that are based on satisfiability of Boolean formulas in conjunctive normal form validate designs by forcing bounds on the range of variables of a design. Their primary drawback is that they are limited to designs of relatively low complexity. They translate a design expressed in an imperative or a declarative high-level description language to a Boolean formula. There are three limiting aspects of translating high-level designs to conjunctive normal form. (1.) A small increase in the bound on variable ranges can cause a large increase in the size of the translated formula. For example, for an undirected

seven-node tree the translation produced one million variables and five million clauses. (2.) Boolean satisfiability solvers are restricted to using optimizations that apply at the level of conjunctive normal form formulas. Finally, (3.) the Boolean formulas often need to be regenerated with higher bounds to ensure the correctness of the translation.

This dissertation proposes the use of sequential circuits, Boolean formulas with memory elements and hierarchical structures, and sequential circuit solvers to validate high-level designs. (1.) Sequential circuits are much more succinct than the pure combinational conjunctive normal form formulas and preserve the high-level structure of the design. (2.) Encoding the problem as a sequential circuit enables the use of a number of powerful automated analysis techniques that have no counterparts for conjunctive normal form formulas. This dissertation applies sequential analysis to both declarative and imperative designs. The results show that it can validate designs with bounds that are orders of magnitude larger than those achievable by the state of the art techniques based on conjunctive normal form analysis.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

As computing systems and their applications steadily grow in complexity and size, designing such systems manually becomes more and more error-prone. Bugs and critical errors become extremely expensive to fix or compensate if discovered in late stages of the design cycle or even after shipping. A report by the National Institute of Standards and Technology (NIST) estimates that software failures cost the US economy about $60 billions every year [1]. Empirically, the effort spent on verifying computing systems constitutes half of the time spent in the design cycle of such systems [2, 3]. Interest in automating validation techniques are thus important to reduce the expense of design bugs.

The question of what is considered a bug or a design flaw comes to mind. The design cycle starts with specifications that describe the functionality and the behavior of a system. These specifications are quantitatively captured with a high-level description language. The implementation and design of the system is implemented usually in another high-level description language. The design and implementation pass through many steps of automatic and manual synthesis and optimizations to end in an executable program in the case of software or a silicon chip in the case of hardware. A certain behavior of a system is considered a bug or a flaw in the design if it does not match the specifications.

Automated techniques in the form of computer-aided design (CAD) tools manage

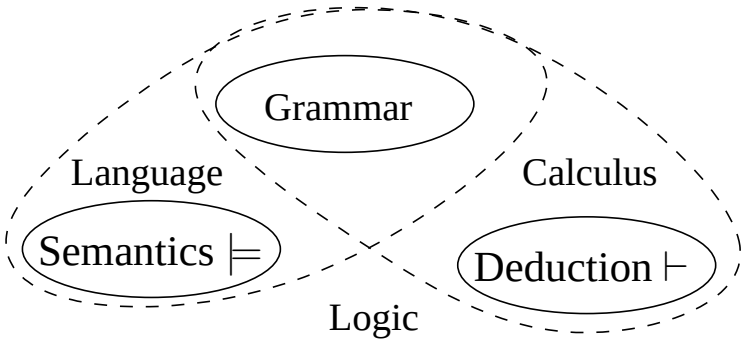


Figure 1.1: Logical system outline.

and perform many of the translation steps from the high-level descriptions to the end product. They check the correctness of the design by checking whether the specifications hold as properties of the design. They also check the correctness of the translation steps to gain confidence that the input and the output of a compiler are, for example, functionally equivalent.

1.1 Logics and languages

High-level description languages are part of logical systems (we will refer to them simply as logics). Logic was the subject of study of Church, Tarski, G*ö*del, Hilbert and others in the early 1900's. The infrastructure of their work was laid down by Russell and Frege who built formal systems in which to develop mathematics. This work in turn built on the work of Cantor on set theory and that of Boole, Pierce, and Schr*ö*der and others on logic [4].

A language as shown in Figure 1.1 consists of a grammar that formally defines a set of symbols as its alphabet and a set of rules to form the strings or terms and the well formed sentences in the language.

The meaning of the language is defined in its semantics. The semantics is a system of interpretation that assigns a meaning to every term in the language. The semantics and the grammar formally define a language. Languages are the basic topic of study in the

branch of logic known as model theory. The key concept in an interpretation system is the logical consequence and is often denoted as $\models$.

A logic is a language with a system of deduction for its grammar [5]. Some formulas of the logic are designated as axioms and deduction rules are set up to allow syntactic moves from one formula to the other. A grammar and a deduction system form a calculus and this is the basic topic of study in the branch of logic called proof theory [6]. The key concept in a deduction system is deducibility and is often denoted with $\vdash$.

A logic is said to be sound if $\vdash \subseteq \models$, or all statements that can be proved with deduction rules are semantically true. If $\models \subseteq \vdash$ then it is called complete, or all statements that are semantically true can be proved with deduction rules. G*ö*del's theorem of completeness proves that the pure first order logic of predicates is in fact sound and complete.

In this dissertation we limit our attention to the semantical analysis of logics rather than the proof theory analysis.

1.2 Dynamic analysis

The prevalent traditional approach to verification is through testing and simulation. The design is tested against a reference model that captures its behavior and presented with sets of concrete inputs that are generated either randomly or with directed methods. Thus they are referred to as dynamic analysis techniques. The use of dynamic techniques requires the creation of a set of concrete inputs that is large enough to expose design problems. It also requires the determination of the correct behavior of the system subject to these inputs, so that it can compare the expected behavior against the actual behavior of the system.

Random dynamic analysis techniques generate concrete inputs and apply them as valuations for the free parameters of a system and monitor the behavior of the system under test [7, 8]. Directed test generation techniques make use of several heuristics to guide the concrete input generation. Constraint based test generation uses known restrictions to the parameters of a system to avoid generating illegal inputs. Coverage based test generation

makes use of previous concrete inputs that were applied to the system and tries to generate concrete inputs that excite different behaviors of the system, thus resulting in higher coverage.

Other forms of dynamic analysis make use of predetermined test cases in the form of regression suites. A regression suite is a collection of test cases that capture a history of interesting cases as well as discovered and fixed flaws. Each test case is associated with a golden model that represents the desired behavior of the system. A system is checked against its regression suite whenever a significant change is applied to it in the hope of preserving robustness.

In short, dynamic techniques are scalable and can be applied to systems of any size. However, they consume lots of time, and are incomplete and unable to prove the absence of errors since they explore only some of the possible behaviors and scenarios of a computing system [4].

1.2.1 Static analysis

Static analysis, also referred to as formal verification, refers to the use of rigorous mathematical techniques to analyze the description and prove the correctness of computing systems. These techniques perform an exhaustive exploration of all possible behaviors of a system and check for its correctness. Static verification techniques fall under either (1.) automatic algorithmic verification techniques or (2.) deductive reasoning techniques also known as theorem proving. In this dissertation we present novel algorithms and techniques that improve automatic verification techniques.

Static verification techniques check whether certain properties or specifications hold for the description of a design. They require the users to provide formal descriptions of the design, its specifications and its properties. They suffer from the well known *state explosion* problem. They usually require exponential resources with respect to the size of the design. For example, symbolic reachability algorithms address a PSPACE-complete problem and

are limited to designs with less than a thousand memory elements despite all advances in recent technologies [9]. Other techniques such as induction [10] and bounded model checking (BMC) [11] solve a simpler NP-complete problem, and thus are applicable to larger designs, however, they often return inconclusive results.

Other techniques that address the complexity problem are based either on small model restricted languages [12] or on abstraction [13]. Most of the abstraction techniques require dedicated manual effort or happen at the low level Boolean representation of the design where many high level design insights are lost [14]. Other techniques such as predicate abstraction [15, 16, 17] use semantic hints and structural heuristics and can be automated.

These techniques are prominently used in the industry to complete proofs as well as to detect design bugs. However, due to their incompleteness, the designer or verification engineer still has to decide how much more resources to invest before gaining enough confidence in the correctness of the design.

1.3 Model checkers

Model checkers constitute a particular class of automatic static verification techniques that explore a combinational space looking for a model that satisfies a property of a design. Static verification techniques based on model checking are embedded in many design tools that are gaining prominence in the last few years. The Alloy Analyzer [18, 19], UCLID [20], MONA [21], VIS [22], Metropolis [23], SMV [9], SixthSense [24], and CBMC [25] are examples of such tools that allow expressing designs formally, as well as checking their correctness to detect crucial flaws that, if not corrected, could lead to failures.

1.3.1 Alloy Analyzer

The Alloy tool set allows the user to formulate his designs in the Alloy language, which is a declarative first-order logic with transitive closure based on relations. The Alloy Analyzer takes the Alloy formula and a bound called scope, a limit on the universe of discourse, and

produces a propositional formula in *conjunctive normal form (CNF)* that is satisfiable if and only if the Alloy formula has a model within the given scope. The Alloy Analyzer then calls an off-the-shelf *satisfiability (SAT) solver* [26, 27, 28] to solve the CNF formula.

1.3.2 UCLID

The UCLID checker is a term-level bounded model checker. It takes designs expressed in Standard ML (SML), which allows a limited form of quantification with integer, Boolean and functional variables. It uses SAT solvers [26, 28], binary decision diagrams (BDD) [29], and a set of abstraction techniques [30] to check the designs. It can handle theories of uninterpreted functions, equality and integer linear arithmetic.

1.3.3 MONA

The name MONA comes from the monadic second order logic theory. In MONA, the user expresses things like parse tree constraints, temporal properties of reactive systems, or search patterns in the weak second-order theory of one or two successors (WS1S). The MONA tool translates the formulas to finite state automata using BDDs and then analyzes the automata to either prove the claimed properties or provide a counter-example [21].

1.3.4 CBMC

CBMC [25] is a bounded model checker for imperative ANSI-C programs that checks for properties such as pointer safety and within-bound array access as well as user assert statements. Given an ANSI-C program and a bound on the range of variables therein, CBMC computes a Boolean formula that asserts the desired properties in the program. It does that by unwinding the transitions and states of a complex state machine that describes the program and its properties into a Boolean formula in CNF and checks the formula using a SAT procedure [27, 31, 32] to look for a counter-example.

These tools take as input a high-level formal description of the design and its properties and synthesize the problem into either a finite automaton or a Boolean satisfiability problem and then try to solve the problem with back-end Boolean solvers such as BDD based symbolic solvers or SAT solvers. During this process, many high-level design aspects that can be instrumental in verification, such as hierarchical boundaries, internal semantical equivalences, and implicative invariants, are lost.

1.4 Limitations of direct translations to SAT

While recent advances in SAT have enabled tools like the Alloy Analyzer and CBMC to check designs of real systems, these designs often need to be partial, leaving out important functionality aspects of the systems, to enable the analysis to complete. Moreover, the analysis is typically bound to relatively small limits, e.g., fewer than 10 entities in a file system with the Alloy Analyzer.

There are three limiting aspects of translating high-level designs to SAT.

Disadvantage 1 The translation to CNF depends on the bounds; a small increase in the bound on variable ranges can cause a large increase in the size of the translated CNF formula due to unwinding loop and recursion structures in programs, or eliminating quantifiers and unrolling transitive closure in declarative logics, for example, for an undirected seven-node tree the translation from Alloy to CNF generates a formula with over one million variables and five million clauses.

Disadvantage 2 The SAT solver is restricted to using optimizations, such as symmetry breaking [33] and observability don't cares (ODC) [34], that apply at the level of CNF formulas. However these optimizations usually aim at increasing the speed of the solver and often result in larger formulas as they add literals and clauses to the CNF formula to encode symmetry and ODC optimizations [35]. Often times when the analyzer successfully generates a large CNF formula, the underlying solver

requires intractable resources.

Disadvantage 3 Often times the CNF formula generated needs to be regenerated with higher bounds in case the unwinding bounds were not large enough for the loops to complete as is the case with CBMC. Note that multiple bounds exist and they need not be all increased during one iteration.

To extend the applicability of static analysis to a wider class of designs and programs as well as to check more sophisticated properties and gain more confidence in the results, we need to scale the analysis to significantly larger bounds; limits on the range of design and program variables.

1.5 Advantages of sequential encoding

This dissertation addresses the complexity of the verification problem by developing analysis techniques that make use of high-level design structure and leverage their power in Boolean solvers.

The limitations of the CNF encoding motivated us to develop new algorithms to encode logics and programs into sequential circuits and decide them using a sequential circuit solver.

We formally define sequential circuits in Section 2.3; for now a sequential circuit can be viewed as a restricted C++ program, specifically a multi-threaded program in which all variables are either integers, whose range is statically bounded, or Boolean-valued, and dynamic allocation is forbidden [36].

Given a design with a property and a bound, we automatically derive a sequential circuit and a Boolean variable therein that serves as an *invariant*, i.e., the variable is stuck to true if and only if the property of the design is valid within the bound.

There are two key advantages to compiling designs into sequential circuits rather than CNF formulas:

Advantage 1 Sequential encodings are much more succinct than pure combinational SAT formulas. In cases, SAT encoding algorithms produce a data structure that uses several orders of magnitude more memory to represent.

Advantage 2 Casting the decision problem for a property of a design as an invariant check on a sequential circuit allows us to make use of a number of powerful automated analysis techniques that we discussed in Section 2.4 and that have no counterpart in CNF analysis.

Intuitively, Advantage 1 holds because sequential circuits are imperative and state-holding while CNF formulas are declarative and state-free. For example, sequential circuits can naturally represent the execution of quantifiers and sequential loops without the need for unrolling them. Moreover, sequential circuits can store and reuse intermediate results in local variables.

We justify Advantage 2 in Section 2.4 and give examples illustrating some automatic analysis techniques that apply to sequential circuits.

We use SixthSense, a tool developed at IBM, to automatically check invariants in sequential circuits. SixthSense reads designs expressed in the very high speed integrated circuit hardware description language (VHDL) [37] as well as several other sequential circuit description languages. Sequential circuits, described as restricted C++ programs in the preceding paragraph, can be efficiently translated into VHDL using inlining [38, 36].

SixthSense is built upon a transformation-based verification (TBV) [39] framework that encompasses reduction and abstraction techniques such as retiming [39], redundancy removal [40, 41, 42, 43], logic rewriting [44], interpolation [45], and localization [46]. It operates on sequential circuits; Boolean netlists with memory elements, and iteratively and synergistically calls numerous transformation and abstraction algorithms. These algorithms simplify and decompose complex problems until they become tractable for decision techniques such as bounded model checking, circuit SAT solving, target enlargement, and semi-formal search [47, 26, 48, 49, 24].

1.6 Contributions

This dissertation presents SERA and SEBAC, two novel algorithms to encode designs expressed in first order relational logic and the C programming language respectively into sequential circuits. Then we apply TBV to check the properties therein.

1.6.1 Sequential encoding for declarative program analysis

SERA [50] compiles an Alloy relational logic formula for a given scope to a sequential circuit. Our experiments show that SERA, used in conjunction with a sequential circuit analyzer, can check formulas for scopes that are an order of magnitude higher than those feasible with the Alloy Analyzer. SERA also exploits high-level insight from the Alloy formula to reduce the number of variables needed in the sequential circuit and to embed invariants as implicative structures where it is easy for the back end tool, SixthSense, to detect them.

1.6.2 Sequential encoding for imperative program analysis

We enabled a software static analysis flow from SPARK [51], a fully automated hardware high-level synthesis tool that translates C programs to sequential circuits. SPARK has not been used for verification purposes before. Sequential encoding for bounded ANSI-C programs (SEBAC) automatically encodes an ANSI-C program with a bound on the ranges of its variables into a sequential circuit with an invariant therein. The SEBAC encoding is more efficient for verification purposes than the SPARK encoding as the latter targets optimizations such as time multiplexing and circuit area reduction on the expense of increasing the number of variables in the circuit.

1.6.3 Experimental evaluation of TBV for program analysis

We evaluate TBV analysis of Alloy designs and C programs. We compare SERA coupled with a TBV solver against the Alloy Analyzer coupled with SAT. We also compare SPARK and SEBAC coupled with a TBV solver against CBMC coupled with SAT.

1.7 Summary and Organization

In this dissertation we present novel static analysis techniques for verifying declarative relational designs and imperative programs. We introduce the use of sequential circuits instead of pure combinational Boolean formulas to encode bounded ANSI-C programs and Alloy designs and thus we enable the use of sequential solvers with reduction potentials that have no counterparts in combinational solvers.

As our results show, sequential analysis techniques scale to bounds that are orders of magnitude higher than those achievable with CNF and SAT techniques.

The rest of this dissertation is structured as follows. Chapter 2 is a review of basic definitions and concepts related to Boolean formulas, sequential circuits, SAT, and TBV. SERA is introduced, formalized, and evaluated in Chapter 3. SEBAC is introduced, formalized, and evaluated in Chapter 4. Finally, Chapter 5 discusses the contributions of this dissertation and suggests extensions and future work.

SERA was presented in the International Conference on Software Engineering 2007 in Minneapolis MN, and SEBAC will appear in the Automated Software Engineering Conference 2007 in Atlanta GA.

Chapter 2

Basic Concepts and Definitions

In this chapter we introduce key concepts such as CNF Boolean formulas, SAT solvers, sequential circuits, and TBV solvers. Boolean functions express a class of problems with a certain expressiveness and difficulty [52]. They may be represented in different ways such as formulas, netlists, circuits, or truth tables, but they all share the same underlying semantics:

- the domain $\mathbb{B} = \{\text{true}, \text{false}\}$
- a finite set of variables $\mathcal{V}$ of type $\mathbb{B}$
- a *valuation* which assigns a value from $\mathbb{B}$ to each of the variables in $\mathcal{V}$.

Definition 1 (Boolean function)**.** A *Boolean function* is a mapping $f : \mathbb{B}^n \mapsto \mathbb{B}$ where n is the number of variables in $\mathcal{V}$.

Definition 2 (Valuation)**.** A *valuation* is a mapping $\sigma : \mathcal{V} \mapsto \mathbb{B}$ which assigns each $x \in \mathcal{V}$ to true or false.

Definition 3 (Validity, Satisfiability)**.** A Boolean function is called *valid* if it results in true for all valuations. The function is called *satisfiable* if there exists at least one valuation which results in the value true.

Definition 4 (Boolean formula)**.** Given the set of atomic formulas $x_i \in \mathcal{V}$, the set of unary operators $\{\neg, \ldots\}$, and the set of binary operators $\{\wedge, \vee, \ldots\}$, *Boolean formulas* are inductively defined as follows.

1. All atomic formulas are formulas.
2. If Φ is a formulas then $\circ\Phi$ is a formula where $\circ$ is a unary operator.
3. If Φ and Ψ are formulas then $\Phi \circ \Psi$ is a formula where $\circ$ is a binary operator.

Nothing else is a formula except as required by conditions 1—3 above.

The semantics of the operators are defined as usual in the form of truth tables where $\neg, \wedge, \vee$ describe negation, conjunction and disjunction respectively. Other symbols such as $\text{true}, \text{false}, \implies$ (implies), $\oplus$(xor), ... will be used to express their usual meaning keeping in mind they are abbreviations that can expressed in terms of the logically complete set $\{\neg, \wedge\}$.

2.1 CNF

A Boolean formula is said to be in CNF when it is a conjunction of CNF clauses where a CNF clause is a disjunction of literals. It is similar to the canonical product of sums form [53, Chapter 4]. Next, we formally define literals, clauses, and CNF formulas.

Definition 5 (Literal)**.** A *literal* is an expression of the form x or $\neg x$ where $x \in \mathcal{V}$.

Definition 6 (CNF Clause)**.** A *CNF clause* is a finite expression of the form $l_1 \vee l_2 \vee \ldots \vee l_n$ where each l_i is a literal.

Definition 7 (CNF form)**.** A formula is in *conjunctive normal form* if it was of the form $C_1 \wedge C_2 \wedge \ldots \wedge C_n$ where each C_i is a CNF clause.

In short a CNF formula is a Boolean formula with two levels of logical hierarchy and can be expressed briefly as $\bigwedge_{i=1}^{n} \bigvee_{j=1}^{m_i} l_{ij}$ where n is the number of clauses, m_i is the

number of variables in clause i, and l_{ij} is a literal expressing either the variable indexed by j in clause i or its negation.

We list the following property of CNF [54].

Corollary 1. Every Boolean formula Φ can be mapped to a formula in CNF form Ψ in linear time such that Φ is satisfiable iff Ψ is satisfiable.

2.2 SAT

Definition 8 (Decision procedure)**.** We write $\models_L (\not\models_L)\Phi$ to mean that Φ is valid (invalid) in logic L. A decision procedure d for logic L is a total function from formulas in L to $\mathbb{B}$ that has the properties of soundness, $d(\Phi) =$ true implies $L \models \Phi$, and completeness $L \models f$ implies $d(f) =$ true.

A Boolean formula Φ is said to be satisfiable if there is a valuation σ to its Boolean variables such that the formula evaluates to true when σ is substituted into the variables of Φ. A Boolean formula is valid if it is true for all its possible valuations. Consequently a Boolean formula is unsatisfiable if and only if its negation is valid. A Boolean formula is thus called satisfiable if a decision procedure $d(\Phi)$ returns true. The satisfiability problem (SAT) of a Boolean formula Φ is to determine a satisfying assignment for Φ. This is an NP-complete problem and many techniques and solvers address it with heuristics that can find satisfying assignments fast when they exist [26, 27, 28].

2.3 Sequential circuits

Definition 9 (Sequential circuit)**.** A *sequential circuit* is a tuple $\big((V, E), G, O\big)$. The pair (V, E) represents a directed graph on vertices V and edges $E \subseteq V \times V$ where E is a totally ordered relation. The function $G : V \mapsto types$ maps vertices to $types$. There are three disjoint types: *primary inputs*, *bit-registers* (which we often simply refer to as *registers*),

and logical *gates*. Registers have designated *initial values*, as well as *next-state functions*. Gates describe logical functions such as the conjunction or disjunction of other vertices. A subset O of V is specified as the *primary outputs* of V. We will denote the set of primary input variables by I, and the set of bit-register variables by R.

Definition 10 (Fanins)**.** We define the direct *fanin*s of a gate u to be $\{v : (v, u) \in E\}$ the set of source vertices connected to u in E. We call the *support* of u $\{v : (v \in I \vee v \in R) \wedge (v, u) \in *E\}$ all source vertices in R or I that are connected to u with $*E$, the transitive closure of E.

For ease of exposition, we restrict gates to have 2 fanins, and compute the NAND function; since NAND is functionally complete, this is not a limitation. For the sequential circuit to be syntactically well-formed, vertices in I should have no fanins, vertices in R should have 2 fanins (the next-state function and the initial-value function of that register), gates should have two fanins, and every cycle in the sequential circuit should contain at least one vertex from R. The initial-value functions of R shall have no registers in their support. All sequential circuits we consider will be well-formed.

2.3.1 Semantics of sequential circuits

Definition 11 (State)**.** A *state* is a Boolean valuation to vertices in R.

Definition 12 (Trace)**.** A *trace* is a mapping $t : V \times \mathbb{N} \mapsto \mathbb{B}$ that assigns a valuation to all vertices in V across time *steps* denoted as indexes from $\mathbb{N}$. The mapping must be consistent with E and G as follows. Term u_j denotes the source vertex of the j-th incoming edge to

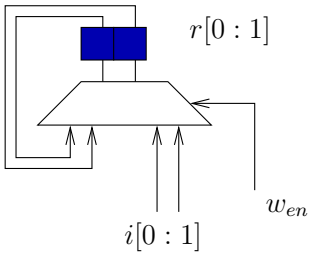
v, implying that $(u_j, v) \in E$. The value of gate v at time i in trace t is denoted by $t(v, i)$.

$$t(v,i) = \begin{cases} s_v^i & : v \in I \text{ with sampled value } s_v^i \\ t(u_2, i-1) & : v \in R, i > 0, u_2 := \text{ next-state of } v \\ t(u_1(v), 0) & : v \in R, i = 0, u_1 := \text{ initial-state of } v \\ G_v\big(t(u_1,i), ..., t(u_n,i)\big) & : v \text{ is a combinational gate with function } G_v \end{cases}$$

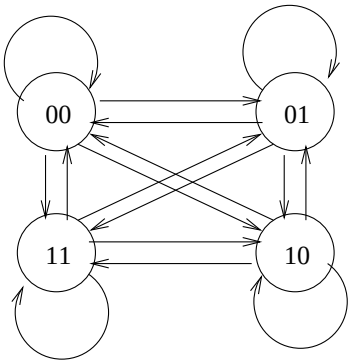
The semantics of a sequential circuit are defined with respect to semantical traces. Given an input valuation sequence and an initial state, the resulting trace is a sequence of Boolean valuations to all vertices in V which is consistent with the Boolean functions at the gates. We will refer to the transition from one valuation to the next as a *step*. A node in the circuit is justifiable if there is an input sequence which when applied to an initial state will result in that node taking value true. A node in the circuit is valid if its negation is not justifiable. We will refer to targets and invariants in the circuit; these are simply vertices in the circuit whose justifiability and validity is of interest respectively.

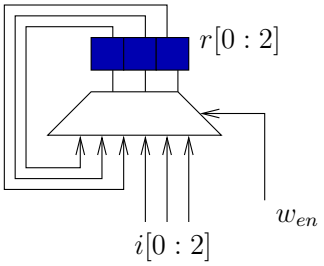
2.3.2 Comparison with finite state machines

A sequential circuit can naturally be associated with a finite state machine (FSM), which is a graph on the states. However, the circuit is very different from its FSM; among other differences, it is exponentially more succinct in almost all cases of interest [55]. Figure 2.1(a) shows a 2-bit write enabled buffer which uses a multiplexer enabled by w_{en} to update its register bits r with values in inputs i when w_{en} is 1 and retains its state otherwise. It also shows the corresponding FSM with no labels on transitions for clarity of exposition purposes. Figure 2.1(b) shows a 3-bit version of the same circuit with the corresponding FSM. Only the arcs corresponding to state 000 are shown for clarity of exposition.



(a) 2−bit write enabled buffer





(b) 3−bits write enabled buffer

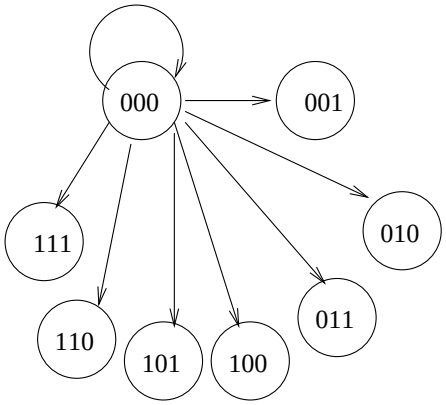


Figure 2.1: Sequential circuits and FSMs with no labels on transitions and omitted transitions for the FSM in (b) for clarity.

2.3.3 Illustrating sequential circuits with C++ classes

An alternative way to understand a sequential circuit is to think of it as a set of communicating and concurrent threads. A sequential circuit is easily understood as a C++ object with Boolean variables describing its states and member functions describing its output and computation.

```
1  class SequentialCircuit {
2    Boolean registerVariables[];
3    Boolean done;
4
5    void initialState( Boolean inputs[]);
6    void nextState( Boolean inputs[]);
7    Boolean outputFunction(Boolean inputs[]);
8
9    Boolean executeCircuit(Boolean inputs[]){
10      initialState(inputs);
11      while(!done)
12        nextState(inputs);
13      return outputFunction(inputs);
14    }
15 }
```

The class `SequentialCircuit` has a set of register variables that describe its state along with the `done` variable that denotes the completion of the computation. It also accepts primary input variables as parameters to its functions.

The `initialState` function assigns the initial state of the sequential circuit by assigning values to its register variables that are restricted to be combinational functions of the `inputs`. The `nextState` function updates the state of the sequential circuit by assigning values to the register variables that are combinational functions of the `inputs` and the `registerVariables` themselves. The `nextState` function performs the computational function of the sequential circuit and assigns `done` to a true value once it completes the

Table 2.1: TBV algorithms used in the experiments of this dissertation.

COM	merges functionally equivalent logical gates using low complexity analysis [56]. COM only uses combinational analysis and hence solves NP-complete subproblems
EQV	makes intelligent guesses on sequential equivalency [42] and performs computationally expensive checks (PSPACE-complete problems) that allow significant merging of logic when they pass. It also exploits structural symmetry detection.
RET	reduces the number of registers by shifting them across combinational gates [39].
BMC	performs an exhaustive BDD or SAT based analysis within a given limit on space or time resources [47].
CMSA	performs compositional minimization [57] by isolating components in the sequential circuit and detecting equivalent states within the component.
BIG	replaces a target by a re-encoding, i.e., a set of states which will hit that target within k time steps [24].
LOC	is a localization based *abstraction* and refinement engine. It overapproximates the target by replacing the gates on a boundary with primary inputs [24].
CUT	replaces a set of gates with a simpler yet equivalent sequential circuit; reduces input count via defining inputs as functions of each other [24].
SCH	conducts a semi-formal search for a target using a hybrid approach of random simulation, symbolic simulation, and induction [48, 56].

computation. The `outputFunction` returns the result of the computation. Note that a sequential circuit may have more than one output function where some of these functions denote the validity of the output. The function `executeCircuit` calls the `initialState` function to initialize it and then calling `nextState` to perform the computation. It then returns the output of the computation via calling the `outputFunction`. The `while` in `executeCircuit` models time where each iteration is a step.

2.4 Transformation-based verification

The concept of TBV has been proposed to apply various transformation techniques and algorithms iteratively to simplify and reduce a large problem into sufficiently small problems

that may be discharged easier. TBV applies simplification and reduction techniques iteratively on a sequential circuit to reduce the verification complexity by reducing the number of logical operations, register variables, and inputs in the circuit. Then it attempts to solve the problem via decision techniques such as bounded model checkers, circuit SAT solvers, or semi-formal searches. The decision techniques aim to find a satisfying trace that is an assignment to the initial value functions of the register variables and a sequence of input valuations that result in asserting the output function to a true value at the last step of the trace.

There is an arsenal of techniques for automatically optimizing sequential circuits; examples include variable minimization via retiming [39], common subexpression extraction [58], and exploiting reduced observability and controllability at internal components [59, 60]. In Table 2.1, we briefly describe various transforms and comment on their efficiency. We discuss two techniques that were particularly useful in our experiments.

2.4.1 Abstraction

Consider the verification of library code L which uses a sophisticated memory allocator for performance.

Let the library $L*$ be L with L's allocator abstracted to a simpler allocator that nondeterministically selects a block from the set of free blocks. Since the simpler allocator uses nondeterminism, if an invariant holds of $L*$, it holds of L. The simpler allocator in $L*$ makes invariant verification on $L*$ easier than it is on L.

While there exist efficient algorithms for automatically identifying components for abstraction in sequential circuits [61, 24], abstraction for CNF formulas is much harder. This is because there is no structure in a CNF formula to guide the abstraction algorithm—the clauses are unordered. Work to extract some structure from CNF formula [62] is less valuable when the original design is present.

Note that an invariant may fail on $L*$, but hold of L, e.g., L's code makes use of

details from the implementation of the allocator beyond those exported from the abstract interface. The localization based abstraction (LOC) in Table 2.1 will automatically identify a negative as false, and roll back the abstraction [61, 24].

2.4.2 Compositional minimization

Consider the verification of a spanning tree algorithm T which uses a balanced search tree (BST) to manipulate sets.

With respect to its abstract interface, a BST implementation of sets is functionally equivalent to a list implementation of sets. Let spanning tree algorithm T' be T with sets implemented using lists. Because a BST is more complex than a list, verification of T' is easier than verification of T. Since the list and BST representations of sets are equivalent with respect to their abstract interface, an invariant holds of T iff it holds of T'.

There exist several techniques for automatically identifying components and minimizing them in sequential circuits [42, 43] including CMSA [57] from Table 2.1. These techniques are based on the notion of equivalent states [63]. Analogous techniques do not exist for CNF formulas, as there is no notion of state.

2.4.3 TBV flow

In Figure 2.2 we show a sample iterative TBV flow. The netlist `RING` is presented with 1.2×10^5 registers and 2×10^4 inputs and reduced to $4,598$ registers and $8,453$ inputs before passing it to SCH which happens to find a counter-example. The TBV flow starts with a redundancy removal transform (COM-1), then retiming (RET) is applied. The compositional minimization transform (CMSA-1) re-encodes components in the netlist and the result is passed to another redundancy removal pass (EQV-1) to leverage the synergy between the two algorithms. The input reparamatrization transform (CUT) further reduces the netlist and passes it to CMSA, COM, and EQV again. Then the netlist is passed to SCH which finds a counter-example using its semi-formal search algorithm.

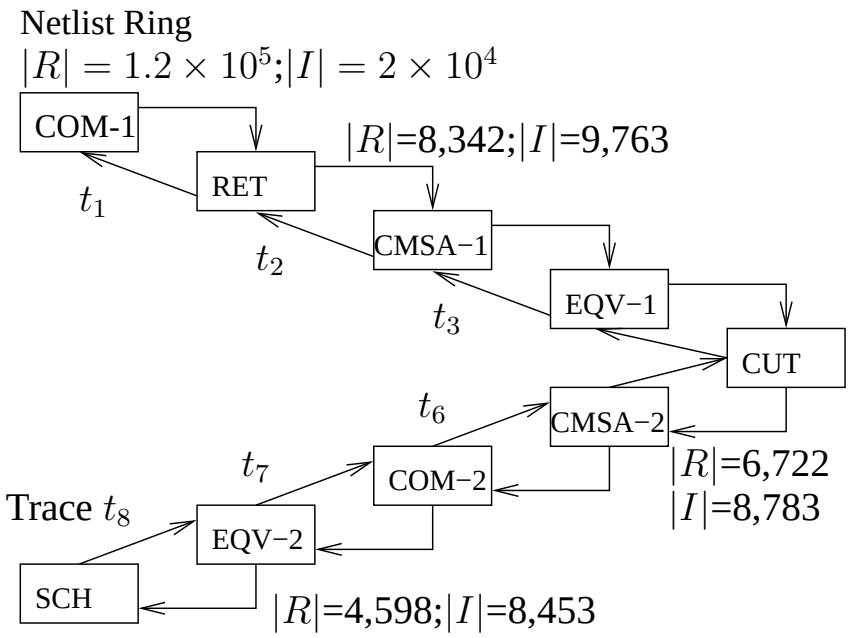


Figure 2.2: Typical TBV transform flow.

SCH finds a counter-example for the netlist presented to it with only $4,598$ registers and $8,453$ inputs in the form of a trace t_8. SCH passes the trace t_8 to the calling transform which is responsible for undoing the effects of its transformation on the This process is called trace lifting and it propagates the trace from one transform to the other until we reach the root transform where the trace t_1 matches the input netlist provided by the user.

Finding well-tuned transform flows is not a trivial task. First there are infinitely many flows to consider. Second, many transforms, such as localization (LOC), are irreversible and thus backtracking is needed. Also, it is difficult to measure the effectiveness of a given transformation. Design size metrics such as the number of inputs, registers and gates are generally good metrics but may be misleading at times since, for example, a transformation may decrease one metric but increase another. Techniques based on a large set of rules that leverage synergy amongst algorithms and that is also based on recorded and evaluated flow sequences automates the process of finding successful flows of transforms [24].

Chapter 3

Sequential Encoding for Declarative Programs

3.1 The Alloy Analyzer

The last few years have seen a new generation of design tools that allow expressing designs formally, as well as checking their correctness to detect crucial flaws that, if not corrected, could lead to failures. The Alloy tool-set is one such design tool that is rapidly gaining prominence [18, 19]. The user formulates his design in the Alloy language, which is a first-order logic with transitive closure based on relations, and checks the correctness properties using the Alloy Analyzer.

The Alloy tool-set has been used successfully to check designs of various applications, such as Microsoft's Common Object Modeling interface for interprocess communication [64], the Intentional Naming System for resource discovery in mobile networks [65], and avionics systems [66], as well as designs of cancer therapy machines [67].

The Alloy language provides a convenient notation based on path expressions and quantifiers, which allow a succinct and intuitive formulation of a range of useful properties, including rich structural properties of software.

Much of Alloy's utility, however, comes from its fully automatic analyzer, which performs a bounded exhaustive analysis using propositional SAT solvers. Given an Alloy formula and a *scope*, i.e., a bound on the universe of discourse, the analyzer translates the Alloy formula into a CNF Boolean formula, and solves it using an off-the-shelf SAT solver [26, 27, 28].

3.2 The case for sequential circuits

The limitations of the CNF encoding discussed in Section 1.4 limit the Alloy analysis to relatively small scopes, e.g., fewer than 10 entities in a file system. These limitations are mainly due to the necessity to eliminate quantifiers and unroll transitive closure operators in order to translate Alloy formulas into CNF, e.g., for an undirected graph description of a tree bounded to seven ndoes the translation generates a formula with over 1 million variables and 5 million clauses.

To increase Alloy's applicability to a wider class of systems as well as to checking more sophisticated properties of designs and gaining more confidence in the results, we need to scale Alloy's analysis to significantly larger scopes.

The limitations of the CNF encoding motivated us to develop sequential encoding for relational analysis (SERA), an algorithm which encodes Alloy formulas as sequential circuits and decides them using a sequential circuit solver.

Given an Alloy formula and a scope, SERA automatically derives a sequential circuit and a Boolean variable therein that serves as an invariant, i.e., the variable can be set to true if and only if the Alloy formula is satisfiable within the scope. (For ease of exposition, we will sometimes refer to the output of SERA as a circuit, with the invariant being implicit.)

We use SixthSense [24, 57], a tool developed at IBM, to automatically check invariants on sequential circuits. SixthSense reads designs expressed in the VHDL design language [37]. Sequential circuits, as described in the preceding paragraph, can be effi-

ciently translated into VHDL using inlining [38, 36].

In SERA we make the following key contributions:

1. **New encoding for Alloy:** We propose SERA, an algorithm that encodes an Alloy formula and a scope as a sequential circuit. We tailor the sequential circuits in order to reduce the number of Boolean variables, inputs and registers, needed to encode the design and its specifications.

2. **Relational analysis:** Our encodings enable the use of sequential circuit verification including many powerful reduction techniques for relational model checking.

The rest of this chapter is structured as follows. We first visit an illustrative Alloy example in Section 3.3. Section 3.4 presents a formalization of Alloy and describes the Alloy Analyzer encoding. The construction of the SERA sequential circuit is presented in Section 3.5. The encoding algorithm is described in Section 3.6, its correctness is proven in Section 3.7, and the approach is evaluated in Section 3.8. Finally, the Chapter concludes in Section 3.9.

3.3 Alloy example

We illustrate the key Alloy constructs through an example; more details are available elsewhere [18]. Consider a *tree*, i.e., a connected, acyclic, undirected graph. There are various equivalent ways of defining trees. We take five textbook definitions [54], model them in Alloy, and check their equivalence using the Alloy Analyzer.

Let $G = (V, E)$ be an undirected graph, where V is a set of vertices and E is a binary relation on V. The following statements are equivalent for non-empty graphs:

1. G is a tree.

2. G is connected, but removing any edge from E results in a disconnected graph.

3. G is connected, and $|E| = |V| - 1$.

4. G is acyclic, and $|E| = |V| - 1$.

5. G is acyclic, but adding any edge to E results in a graph that has a cycle.

An Alloy model consists of *signature* declarations that introduce basic sets and relations, as well as *formulas* that constrain them. For tree, we declare signature `V` to model vertices and binary relation `E` to model edges as shown on Lines 1 and 2 of the tree integrity example code.

Tree integrity example

```
1  sig V { // V: vertices, E: V <-> V edges
2    E: set V }
3
4  fact UndirectedGraph { // E is symmetric
5    E = ~E }
6
7  fact NonEmpty { // consider non empty graphs
8    #V >= 1 }
9
10  pred InCycle(v: V, c: V -> V) {
11    v in v.c or
12    some v': v.c | v' in v.^( c - (v -> v') - (v' -> v))
13  }
14
15  pred Acyclic() {
16    all v: V | not InCycle(v, E)
17  }
18
19  pred Connected(c: V->V) {
20    all v1, v2 : V | v2 in v1.*c
21  }
22
23  pred Statement1() {
24    Connected(E) and Acyclic()
```

```
}

pred Statement2() {
  // connected, removing an edge makes it disconnected
  Connected(E) and
  all u : V |
    all v : u.E  |
      not Connected( E - (u->v) - (v->u))
}

pred Statement3() { // connected and |E| = |V| - 1
  Connected(E) and #E = #V + #V - 2
}

pred Statement4() { // acyclic and |E| = |V| - 1
  Acyclic() and #E = #V + #V - 2
}

pred Cyclic(c: V->V) {
  some v : V | InCycle(v, c)
}

pred Statement5() {
  // acyclic, but cyclic if any edge is added
  Acyclic()
  all u,v : V |
    (u->v) not in E implies Cyclic(E + (u->v) + (v->u))
}

assert EquivOfTreeDefns {
  Statement1() implies Statement2()
  Statement2() implies Statement3()
  Statement3() implies Statement4()
  Statement4() implies Statement5()
```

```
59    Statement5() implies Statement1()
60  }
61
62  //final check is subject to facts being true
63  check EquivOfTreeDefns for 4
```

Alloy comments are prefixed with `//`. The keyword `set` makes `E` an arbitrary relation. We represent an undirected edge between vertices u and w as a pair of directed edges (u, w) and (w, u). `E` is a symmetric relation, which we express using the transpose operator '`~`' on Line 5.

A *fact* introduces a constraint on the declared sets and relations. that must be satisfied by any *instance* of the model, i.e., any satisfying assignment of values to sets and relations. The fact `NonEmpty` on line 7 uses the cardinality operator # to state there is at least one vertex. This condition is required for equivalence of Statements 1–5.

We express Statement 1 on Line 23 using a *predicate*, i.e., a formula that may have free variables and can be invoked elsewhere.

The operator `and` is logical conjunction; Alloy also provides `or`, `not`, `=>` (implication), and `<=>` (iff). The keywords `all` and `some` respectively represent universal and existential quantification; `in` represents subset (and membership); '`.`' denotes relational product; '`^`' denotes transitive closure, and '`*`' denotes reflexive transitive closure. The expression `v2.^E` on Line 12 thus denotes the set of all vertices reachable from `v2` following edges in `E`, and the predicate `Connected` on Line 19 states that there is a path between any two distinct vertices. The predicate `InCycle` on Line 10 states that a vertex `v` is a part of a cycle according to an edge relation `c` iff there is a self-loop at `v` or `v` has some neighbor `v'` such that even if we remove the edge connecting `v` and `v'`, these two vertices are still connected. The operators '`->`' and '`-`' represent pairing (more generally, Cartesian product) and set difference, respectively.

Statements 2–5 can be defined likewise.

On Line 36 we represent the constraint $|E| = |V| - 1$ using the formula `#E = #V`

`+ #V - 2`, since each undirected edge is represented using two directed edges. On Line 54 we express the equivalence of Statements 1–5 using a chain of implications.

An Alloy *assertion*, on Line 54, introduces a formula that should be checked, in this case whether the equivalence holds. The command `check`, on Line 63, instructs the analyzer to find a counter-example to the given assertion using the specified scope, specifically 4.

Besides `check`, Alloy Analyzer also provides a command `run` that directly finds *instances*, i.e., valuations to `V` and `E` that satisfy a given formula as well as the facts. The user can also choose to enumerate satisfying assignments by selecting an enumerating solver such as Berkmin [27], mChaff [26] and relsat [32].

To check `EquivOfTreeDefns`, the analyzer searches for a counter-example, an instance which satisfies the negation of the assertion while satisfying all the facts. If the analyzer fails to generate a counter-example, the formula is valid with respect to the given scope.

For `EquivOfTreeDefns`, the Alloy Analyzer failed to complete its check for a scope of 7 with a time out period of 14,000 seconds. In contrast, SERA successfully checked the assertion for a scope of 32. Section 3.8.2, specifically Table 3.5, presents detailed results, including those for other Alloy designs.

3.4 Alloy formalisms and encoding

We formally define the Alloy constructs and review the Alloy encoding.

3.4.1 Formal description of Alloy

An Alloy formula is expressed in first order over sets and relations. Alloy treats sets as relations with arity 1 to simplify the semantics of the language. We show a simplified grammar of the Alloy language and we define the corresponding semantics.

Definition 13. An Alloy formula $\mathcal{AF} := \mathcal{R}\ \mathcal{R}^*\ \mathcal{F}$ starts with a set of relation declarations $\mathcal{R}$ followed by a formula $\mathcal{F}$. A relation declaration $\mathcal{R} := V : \mathcal{T}$ is an association between a variable expression V and a type expression $\mathcal{T}$. A type expression $\mathcal{T} := T \mid T \mapsto T \mid T \implies \mathcal{T}$ is either a set denoted by a type T, or a relation between two types, or a function mapping a type to a type expression. The latter allows relations with arities higher than 2. A formula $\mathcal{F} := E \in E \mid \neg\mathcal{F} \mid \mathcal{F} \wedge \mathcal{F} \mid \forall V : T.\mathcal{F} \mid \exists V : T.\mathcal{F}$ is either a membership expression $E \in E$, a negation $\neg\mathcal{F}$, a logical conjunction $\mathcal{F} \wedge \mathcal{F}$, a universal quantifier, or an existential quantifier. An expression $E := E{+}E \mid E\&E \mid E.E \mid *E \mid V$ is either a union $E + E$, an intersection $E\&E$, a relational product $E.E$, a transitive closure $*E$, or a variable V. Finally a variable $V := v \mid V[v]$ is either a scalar variable v or a joint product expression $V[v]$.

Definition 14. An Alloy formula is said to be well formed if it respects the following type rules. The binary operations $(\in, +, \&)$ are allowed only if the two operand subformulas are of the same type. A quantifier $(\forall, \exists)$ is allowed only if the variable V is of the type T. A relational product $A.B$ with $A : S_1 \mapsto T$ and $B : S_2 \mapsto T$ is allowed only if the types S_1 and S_2 are identical. A transitive closure $*E$ is allowed only if E is a binary relation. A joint operation $V[v]$ is allowed only if V was a function from the type of v to another type.

The semantics of the Alloy language are expressed in terms of interpretations $\mathcal{M}$ with Boolean values (true and false) to Boolean Alloy subformulas, and of interpretations $\mathcal{X}$ with relational values to Alloy subformulas expressing sets and relations. The functions $\mathcal{M}$ and $\mathcal{X}$, defining the meaning of the Alloy constructs, take as argument the constituents of the operator, its operands and the type declarations that form an environment for the construct.

Definition 15. We define the semantics of the Alloy constructs with the following recursive equalities. A membership expression $a \in b$ is a Boolean expression with a true interpretation if the interpretation of its left handside operand is a subset of the interpretation of its right handside operand and false otherwise, $\mathcal{M}_e(a \in b) = \mathcal{X}_e[a] \subseteq \mathcal{X}_e[b]$. The logical

Boolean operations can all be defined in terms of negation and conjunction. The meaning is defined recursively such that $\mathcal{M}_e(\neg F) = \neg\mathcal{M}_e(F)$, and $\mathcal{M}_e(F \wedge G) = \mathcal{M}_e(F) \wedge \mathcal{M}_e(G)$. A universal quantifier expression $\forall v : t.F$ is defined in terms of a multiple conjunction of F applied to all possible interpretations of v in t and thus $\mathcal{M}_e(\forall v : t.F) = \bigwedge_{(v \in t)} \mathcal{M}_e(F)$. An existential quantifier $\exists v : t.F$ can be defined in terms of a universal quantifier and negation but we also define it as a multiple disjunction of F applied to all possible interpretations of v in t and thus $\mathcal{M}_e(\exists v : t.F) = \bigvee_{(v \in t)} \mathcal{M}_e(F)$. The union $A + B$ and intersection $A \ \& \ B$ of two relations is defined to be the union and intersection of the interpretations of A and B such that $X_e(A + B) = X_e(A) \cup X_e(B)$ and $X_e(A \& B) = X_e(A) \cap X_e(B)$. The relational product $a.b$ is a navigation operator that concatenates the first column of a to the column of b if the second column of a is identical to the first column of b such that $\mathcal{X}_e(a.b) = \{(x, z).\exists y.(x, y) \in \mathcal{X}_e(a) \wedge (y, z) \in \mathcal{X}_e(b)\}$. The transitive closure $*A$ of a relation is the smallest transitive relation that contains A and can be defined as $\mathcal{X}_e(*A) = \min(\{r : \mathcal{X}_e(r.r) \subseteq \mathcal{X}_e(r) \wedge \mathcal{X}_e(A) \subseteq \mathcal{X}_e(r)\})$.

We are left with the interpretation of types, variables, tuples, and constants. Simply the interpretation of either a type or a variable is its name. A tuple is a sequence of names and a constant is a set of tuples.

3.4.2 Alloy encoding

Logics with transitive closure operators are important as they allow the description of such notions as the a set of nodes reachable from the variables of a program. However, adding transitive closure to even simplified logics make them undecidable. Rabin in [68] showed that monadic second-order theory of trees is decidable, however, if we go beyond trees, undecidability comes immediately [69].

Unfortunately, the Alloy first order logic with transitive closure is not decidable [70]. Alloy Analyzer takes scopes that are bounds on the universe of the declarations in the formula and checks the given formula for all its possible interpretations up to the given bound.

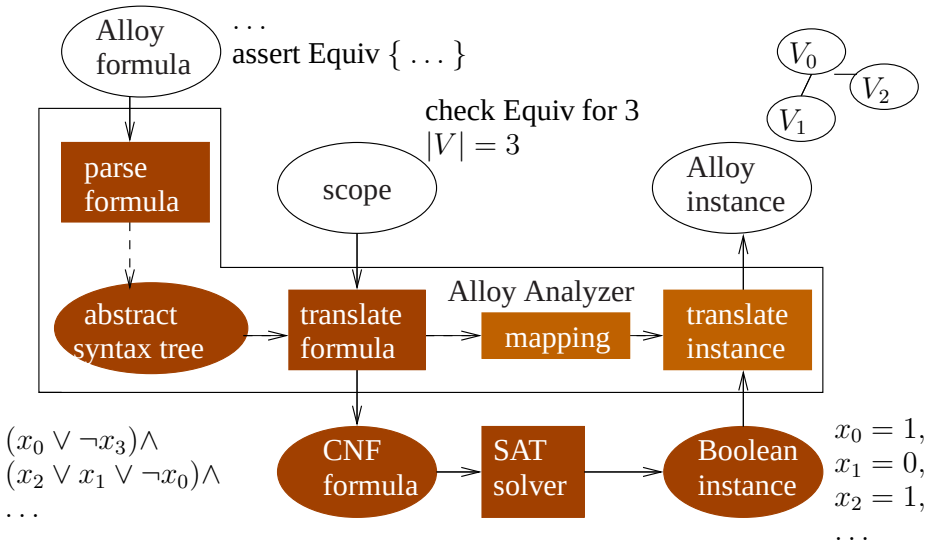


Figure 3.1: Architecture of the Alloy Analyzer.

As shown in Figure 3.1, Alloy Analyzer parses an Alloy formula into an abstract syntax tree (AST) where the nodes are the Alloy constructs described in Section 3.4.1. Then Alloy Analyzer uses the scope to translate the Alloy formula into a Boolean CNF formula as follows.

Alloy Analyzer encodes the problem of checking the validity of an Alloy formula within a given scope into a CNF satisfiability problem; it then calls an off-the-shelf SAT solver to decide the problem [70].

An Alloy relation T is encoded into a bit matrix $\mathcal{T}$. If T relates the i-th object of type A to the j-th object of type B, then $\mathcal{T}$ is formed such that the i and j-th entry of its projection over A and B is set to true, i.e., $\mathcal{T}_{A,B}(i,j) = 1$. The scope limits the range of indices, and thus the matrix is finite.

Membership of a tuple in a relation $\langle v_1, v_2 \rangle \in \mathcal{T}$ is encoded by indexing into the relation with the appropriate indexes $\mathcal{T}_{t:v_1,t:v_2}(v_1, v_2)$. A subset check $R \subseteq \mathcal{T}$ is consequently a conjunction of membership checks $\bigwedge_{\langle v_1,v_2 \rangle \in R} \mathcal{T}_{t:v_1,t:v_2}(v_1, v_2)$. Alloy Analyzer introduces fresh matrices (additional variables) and forces constraints to encode operators.

Intersection is encoded with constraints such as $\big(A \cap B\big)(i,j) = A(i,j) \wedge B(i,j)$, union with $\big(A \cup B\big)(i,j) = A(i,j) \vee B(i,j)$, and negation with $\big(\neg A\big)(i,j) = \neg A(i,j)$. The same applies to the relational product operator $A.B$ and the constraints in that case are $\big(A.B\big)(i,j) = \bigvee_k A(i,k) \wedge B(k,j)$. Transitive closure is encoded as an expansion of the relational product over union up to the scope and thus the corresponding constraints are $(*A)(i,j) = A(i,j) \vee (A.A)(i,j) \vee \ldots \vee A^k(i,j)$. Boolean arithmetic connectives such as conjunction, disjunction, and negation are built on top of their subformulas. A quantifier folds its formula over the range with either conjunction (universal) or disjunction (existential) operations.

Finally, we end up with a Boolean formula P encoding the Alloy predicate to be checked and a conjunction of Boolean constraints $C_1, C_2, \ldots, C_n$ relating the introduced variables to the declared ones. Alloy Analyzer translates the formula $F = \big((C_1 \wedge C_2 \wedge \ldots \wedge C_n) \implies P\big)$ (or its negation in case we are checking for validity) into CNF and passes that to a SAT solver. In case the SAT solver comes back with a Boolean instance satisfying F, then as shown in Figure 3.1 the Alloy Analyzer maps the Boolean instance into an Alloy instance and shows that as a consistency witness in case we are testing the property, or a counter-example in case we were checking the validity of the property. Otherwise, in case no counter-example exists, the Alloy Analyzer concludes the validity of the property within the scope.

We faced three main limitations when we tried to apply SixthSense on a pure combinational circuit translated directly from the CNF formula. First, the Alloy Analyzer often failed to produce the CNF formula due to the large number of variables needed. Second, the combinational nature of the encoding precludes the power of the available sequential transforms in SixthSense, and developing analogous combinational transforms is impossible in some cases. Finally, the huge number of variables presented to the decision algorithms prevented a conclusive result. This motivated the need for a better encoding—specifically, sequential rather than combinational encodings.

Table 3.1: Time steps and Boolean state variables for the sequential SERA components.

Alloy Construct	Validity depth		Number of variables
	Membership	Cardinality	
Signature	0	0	$\lg(n)$
Relation	0	0	n^k
Relational Product	n	n	$\lg(n)$
Transitive Closure	$\lg(n)$	$\lg(n)$	n^2
	Predicate		
Universal Quantifier	1 to n		$\lg(n)$
Existential Quantifier	1 to n		$\lg(n)$

3.5 Construction of SERA(Φ, n)

For the reasons given in Section 1.4 we developed SERA, an algorithm for encoding Alloy formulas into sequential circuits. Given an Alloy formula Φ with a scope n, the SERA algorithm constructs a sequential circuit SERA(Φ, n). The construction proceeds recursively on the abstract syntax directed acyclic graph (DAG) for Φ. At each node in the DAG for Φ we construct a sequential circuit with a special structure for the formula rooted at that node. We refer to each such circuit as a SERA component.

We use SERA components to encode the Alloy design as a sequential circuit. This differs from techniques used in monitor circuits and high performance sequential synthesis [71, 72] since while they match a set of strings to an expression, we produce all matching assignments of a formula within a given scope. A SERA component representing a set or a relation provides an easy implementation of membership and cardinality and allows parallel access to both when the component is in a valid state.

3.5.1 The SERA component

We illustrate the SERA components using C++ classes and objects. For ease of exposition we omit access modifiers and trivial constructors. The abstract class `Component` in Table 3.2(a) describes the generic interface of all SERA components. `Component` inherits from `Thread` to denote that all components run concurrently and its member function

Table 3.2: Pseudo C++ description for abstract component and `sig`.

```
class Component:public Thread{
public:

  bool predicate();
  bool in(iter u);
  bool in(iter u,iter v);
  int card();

  bool memberValid();
  bool cardValid();
  bool predValid();

  void evaluate();
  void terminate();

  void initialState();
  void nextState();
  int depth();};
```

(a) Sequential component interface

```
template<int scope>
class Sig : public Component {
  class iter{/*omitted details*/};
  int size;

  bool in(iter v){ return v < size;};
  int card() { return size;};

  bool memberValid() { return true;};
  bool cardValid() { return true;};

  void initialState() {
    //non-deterministic choice
    size = choose() % scope;};
  void nextState() {
    size = size;};

  int depth(){return 1;};};
```

(b) Sig-set component

`nextState` is the entry point of the thread.

We classify the functions `in`, `card`, and `pred` as output functions. The functions `memberValid`, `cardValid`, and `predValid` are Boolean validity functions and their return values signal whether a value returned by the corresponding output function is valid or not.

1. Function `in` takes index arguments and returns whether the set or relation described by the component contains the variable or tuple denoted by the indices.

2. Function `card` returns the cardinality of the set or the relation described by the component.

3. Function `pred` returns the Boolean value of a predicate if the component corresponds to a Boolean expression.

Depending on the Alloy sub-formula the component corresponds to, some of these functions may never be invoked, and thus may be left unimplemented.

Table 3.3: Pseudo C++ description for relation and set union components.

```
template<class Sig1, class Sig2>
class Relation:public Component{
  class iter{/*omitted details*/};
  Sig1 & V1;
  Sig2 & V2;
  bitMatrix R[Sig1::scope][Sig2::scope];
  bool in(iter v, iter u){
   return R[v][u];};
  int card(){ return countOnes(R);};

  bool memberValid() { return true;};
  bool cardValid() { return true;};

  void initialState(){
   Sig1::iter u;
   Sig2::iter v;
   for (; u<V1.card();i u++)
    for (v.start(); v<V2.card(); v++)
      //non-deterministic choice
      R[u][v]=choose()%2;
  void nextState(){ R = R;};

  int depth(){return 1;};};
```

(a) Arbitrary binary relation component

```
template<class S1, class S2>
class Union : public Component{
  class iter{/*omitted details*/};
  S1 & V1; S2 & V2;

  bool in(iter u){
    return V1.in(u) || V2.in(u);};
  int card(){
    bitVector in;
    for(iter v; v.valid(); v++){
      in[v] = V1.in(v) ||
        V2.in(v);};
    return countOnes(in); };

  bool memberValid(){
    return V1.memberValid() &&
      V2.memberValid();};
  bool cardValid(){
    return memberValid();};

  int depth(){ return
    max(V1.depth(), V2.depth());};};
```

(b) Set union operator component

The component contains references to other components. A function in a component uses the references to execute other components if their output functions are not valid yet and query them once valid. The other data elements of a component constitute its *state*. We refer to the values of the non-reference data elements of a component at a specific step as the state of the component. The `intialState` function initializes the component to its initial state, and the `nextState` function updates the state. Functions `evaluate` and `terminate` are control functions. They start the execution of the component if it was not in a valid or running state, and force it to stop execution if a top hierarchy does not need the result anymore.

We relate the functionality of a SERA component to the semantics of a sequential circuit as described in Definition 3.1. We use these terms to prove SERA's correctness. *Inputs* are arguments passed to a component's functions and non-deterministic assignments

generated by calls to the function `choose` in `initialState` functions.

The class `SERACircuit` inherits from `SequentialCircuit` defined in Section 2.3.3 and implements its functions.

```
1  class SERACircuit : SequentialCircuit {
2    vector<Component> compVec;
3    int depth;
4    void initialState() {
5      depth = 0;
6      for (int i=0; i<compVec.size(); i++)
7        compVec[i].initialState(inputs);
8    }
9    void nextState() {
10     // run all threads concurrently
11     for (int j=0; j<compVec.size(); j++)
12       compVec[j].nextState();
13   }
14   Boolean executeCircuit() {
15     initialState();
16     while ((!compVec[0].predValid()) &&
17            (depth++ < compVec[0].depth())) {
18       nextState();
19       waitForAllThreads();
20     }
21     return compVec[0].pred();
22   }
23 };
```

The vector `compVec` contains all the components SERA generated, and `compVec[0]` is the top level component which corresponds to Φ. The `while` loop in `executeCircuit` models time where each iteration is a step, and the number of steps it takes the loop to terminate is the depth of the circuit. The function `executeCircuit` makes sure to call all `nextState` functions synchronously at every step. The `while` loop spawns all the threads

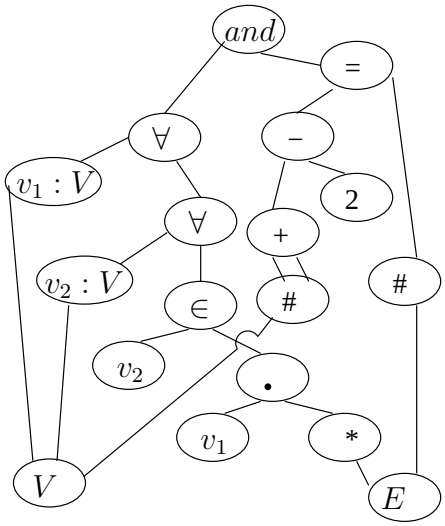


Figure 3.2: `Statement3` predicate diagram.

concurrently and then `SERACircuit` waits for all `nextState` calls to finish. The invariant we need to check is the `pred` function of `compVec[0]`—Φ will be satisfiable in scope n iff `compVec[0].pred()` returns true on termination of `executeCircuit()`.

Tables 3.2, 3.3, and 3.4 show C++ classes describing some of the components corresponding to Alloy constructs. Section 3.6 describes the constructs and their translation in detail. The component's computation is complete when its `predValid` function returns true. Table 3.1 shows the number of state variables as well as an upper bound on the number of steps needed for the computation of an individual component to complete as a function of the scope n. The final depth of the SERA circuit depends on the scope n, and the formula itself and thus it can be computed at compile time via calling the `depth` function of the top level component.

3.5.2 Sequential circuit example

The diagram in Figure 3.2 shows an abstract syntax graph for the `Statement3` predicate. We start at the `and` node and compute the components for the two sub-formulas rooted at

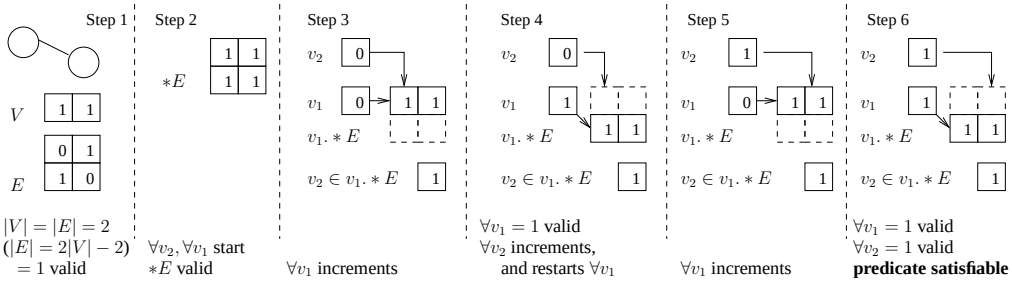


Figure 3.3: SERA execution of a consistent instance of the `Statement3` predicate with a scope of 2.

this node. If a component was previously instantiated, we connect to it appropriately. The code below describes the resulting SERA sequential circuit for the `Statement3` predicate.

`SERA(Statement3, 2)`

```
1  typedef Sig<2> S1;
2  typedef S1::iter S1Var;
3
4  S1 V;
5  Relation<S1,S1> E;
6  TClosure<S1,S1> Et(E);
7  S1Var v2, v1;
8  Product P(v1, Et);
9  Belongs B(v2, P);
10
11  ForAll<S1> A1(v1, V, B);
12  ForAll<S1> A2(v2, V, A1);
13
14  IntPlus P1(V, V);
15  IntMinus M1(P1, 2);
16  IntEqual E1( M1, E);
17  And A3(A2, E1);
```

We pass 2 as a template parameter for `S1` to set its scope at compile time. The constructor of each component initializes its references appropriately and adds itself to the

vector of components. Classes `Sig`, `Relation`, `TClosure`, `Product`, `Belongs`, `ForAll`, `IntPlus`, `IntMinus`, `IntEqual`, and `And` inherit all from `Component` and each implements the Alloy construct as its name suggests.

The 6-step trace in Figure 3.3 shows an execution of the sequential circuit corresponding to `Statement3`. Step 1 shows the graph instance, and the `V` and `E` encodings. `V` is initialized to indicate the existence of both members, and `E` is initialized to indicate the edge in the graph. The membership and cardinality state of `V` and `E` is valid immediately since it corresponds to the initial state. Thus, `#E=#V+#V-2` is true and valid immediately. In Step 2 the quantifiers $\forall v_1$ and $\forall v_2$ are executed as well as the transitive closure on `E`. Since the quantifiers depend on `*E`, their validity has to wait for the `Et` component to signal membership validity. Fortunately for this example, this happens in one step since as we will describe Section 3.5, transitive closure takes $\lg(n)$ steps to complete where n is the scope. In Step 3, the validity of `Et` is propagated to the relational product `P` corresponding to $v_1 . * E$ and `P` is immediately tested for membership of v_2. Since all data is valid, the $\forall v_1$ component updates its predicate state and increments its iterator v_1. The same happens in Step 4, and now $\forall v_1$ completed execution and thus can signal the validity of its predicate. The $\forall v_2$ quantifier updates its predicate state, increments its iterator v_2 and initializes the $\forall v_1$ component to start execution again. Step 5 is similar to Step 3, and Step 6 witnesses the completion of execution of the $\forall v_2$ component. The propositional `and` component has now two valid true inputs so it evaluates to true and thus our predicate can be set to true.

3.6 SERA encoding algorithm

As shown in Figure 3.4, SERA parses the Alloy formula to construct a DAG of signatures, relations and operators with Alloy constructs. SERA recursively traverses the abstract syntax DAG for an Alloy formula Φ with a scope n from its command to its signatures and implicit relations. For each Alloy construct, SERA instantiates its corresponding `Component` object. It composes each component into the desired sequential circuit SERA(Φ, n); the

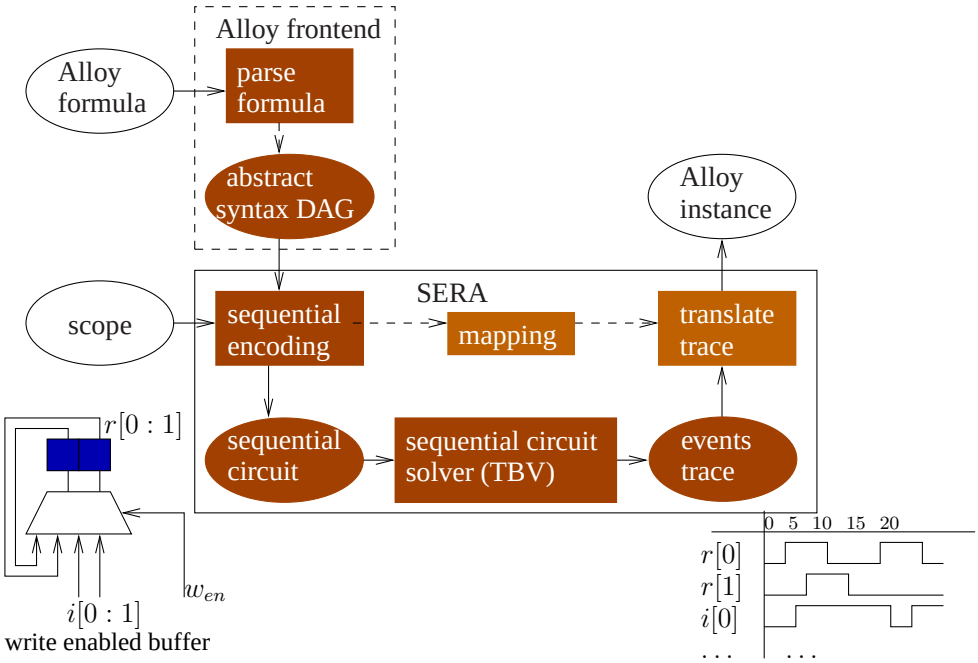


Figure 3.4: Architecture of SERA.

`pred` function of the top level component corresponds to the invariant.

3.6.1 Leaf nodes: signatures and relations

The leaf nodes of the abstract syntax tree of an Alloy formula are signatures and implicit relations and constitute the base case for the SERA(Φ, n) recursive construction. The template class `Sig` in Table 3.2(b) takes a template parameter as its scope and thus the scope is part of the structure of the class. The`initialSate` function uses the return value of a nondeterministic function, `choose`, modulo the scope, to initialize the size. Without loss of generality, the size of the set is enough to represent the set since its members are indexed arbitrarily and any set can be re-indexed appropriately. The functions `card` and `in` are valid immediately and `in` returns true if the index is smaller than the size. The template class `Relation` in Table 3.3(a) implements the binary implicit Alloy relations. A relation declaration takes two set types, `Sig1` and `Sig2` as template parameters and references the

two sets, `V1` and `V2`, it relates. The relation class uses the scope parameters of `Sig1` and `Sig2` to declare the `bitMatrix R` which stores the membership state. The `bitMatrix R` is initialized nondeterministically modulo the cardinality of `V1` and `V2`. The `countOnes` function returns the number of set bits in `R` to compute the cardinality of the relation.

In a trade off between the depth of the circuit and the number of variables, we allow n multiple accesses to the membership functions. This requires additional $\lg(n)$ bookkeeping variables and keeps the sequential depth of all components linear in n.

3.6.2 Internal nodes

The internal nodes of an Alloy formula's abstract syntax DAG correspond to a variety of logical and relational operations, including propositional connectives, quantifiers, relational product, transitive closure, set operations, and arithmetical operators and predicates. We now show in turn how to build the SERA component for each internal node, assuming we have SERA components for all its sub-nodes.

Propositional operators

SERA encodes each propositional operator (`and, or, not, implies, iff`) with a combinational circuit component. The component holds references to its operand components and uses their `pred` and `predValid` functions. We show next a logically complete `Nand` component and the rest of the operators can be described in terms of the `Nand` component.

```
1  class Nand:public Component{
2    Component & F1, & F2;
3
4    bool predicate(){
5      return !(F1.predicate() && F2.predicate());};
6
7    bool predValid(){
```

```
8      return (F1.predValid() && !F1.predicate())||
9        (F2.predValid() && !F2.predicate())||
10        (F1.predValid() && F2.predValid());};
11
12    int depth(){
13      return max(F1.depth(), F2.depth());};};
```

In case of a dominant valid input value, such as a false value in a conjunction, a termination signal may be passed to the component producing the other operands. We call this early termination and later we extend the same concept to the quantifiers.

Quantifiers

Because the scope is finite, we can readily perform quantifier elimination. The universal quantification of θ by x is replaced by the conjunction of θ restricted to each value x can take; for existential quantification, conjunction is replaced by disjunction. The `ForAll` component shown in Table 3.4(a) implements a universal quantifier. It takes a set type, `Sig`, a set, `V`, as its quantification domain, a reference to an iterator, `v`, as the quantified variable, and a reference to the formula component, `F`. It computes conjunction (disjunction) sequentially, and employs an early termination mechanism where the first false (true) value terminates the computation. This mechanism gives us a substantial advantage since we can abort the quantification without having to compute for the whole domain. The quantification component uses the `v` iterator to evaluate `F`, and accumulates its Boolean `valid` and `value` members.

Relational product

Alloy defines two relational product operators: the '`.`' accessor, and the '`[]`' indexer. The latter can be expressed as syntactic sugar of the former. The relational product of two components A and B is implemented by a component that fills in the index of the right

operand B by the concatenation of tuples from the left operand A and the actual index argument to the relational product component.

Transpose

We define a unique variable order to respect the order in which signatures were declared in Φ and we exploit that order to trivialize our type-determination functions. The transpose operator may produce a result which conflicts with the unique variable order. SERA attempts to rewrite the formula in question to normalize transposition. In cases of conflict, such as the `UndirectedGraph` constraint `E =~E`, or in cases of suboptimality introduced in the variable ordering, SERA resorts to adding a redundant variable appropriately. SERA also adds a constraint that indicates the equivalence of the redundant data so that logic optimization techniques can easily exploit the hint.

Transitive closure

Transitive closure in Alloy repeats one or more compositions infinitely many times. SERA's implementation of transitive closure, the `TClosure` template class shown in Table 3.4(b), takes two set types as template parameters and a reference to the component corresponding to the original binary relation. The `nextState` function encodes the transitive closure using *iterative squaring* [55]. This allows us to use only $n \lg(n)$ variables and allows the computation to complete within $\lg(n)$ steps.

Set operations

As shown in Table 3.3(b) the `in` and `card` functions and their validity can be encoded as a combination of the `in` and `memberValid` functions of the operand components. The `Union` class takes two set types and constructs its own appropriate iterator that maps indices correctly in case the types were different. It also takes references to the components representing its operands. The `card` function inlines membership checks in both its operands

and counts the matches in an intermediary bit vector. The rest of the set operators can be easily described in a similar fashion.

Arithmetic predicates

Another source for Boolean values in Alloy is integer arithmetic comparisons, which may involve cardinality values. The Alloy Analyzer uses the scope to allocate a finite number of integers to model the integer space and simplify the arithmetic predicates. We encode arithmetic operators with combinational circuit components in addition to validity propagation of the operand components. Note that all integer valuations, less the cardinality values, are considered valid by default.

3.6.3 Optimizations

Without loss of generality we support the same scope value for all signatures. This allows us to keep our type-determination as simple as checking whether an index lies within a range. We also simplify the counters embedded in the iterators in some components by restricting the scope to be *a power of two*. With this restriction we allow the counters to start at any state and terminate when they reach that state again. The corresponding component can then call the counter cycle state its idle state. The type-determination of a certain index is now simplified to an appropriate Boolean shift operation.

In most cases some SERA components are guaranteed to complete execution before other components even begin. We use this fact to allow memory sharing between non-overlapping components. Note also that we separate variables based on the functions they are used for. This allows substantial *cone of influence* [61] reductions if for example the cardinality of a component is not checked. Furthermore, this introduces redundancy which can be exploited by redundancy removal transforms.

3.6.4 Mapping instances back to Alloy

To map satisfying assignments back to Alloy, we only need to keep track of the initial values assigned to bit-registers in all `Sig`s and implicit relation SERA components. These values are mapped to an instance of Alloy that has a model iff there exists a trace in the sequential circuit that sets the target gate, corresponding to either a negation of a checked assertion or the value of a consistent predicate, to true at the last cycle.

3.7 Correctness of SERA

Intuitively, the SERA encoding works as the translation preserves the semantics of each operation and guards the validity of the valuation with Boolean functions that are guaranteed to evaluate to true after a finite number of steps. We state the correctness of SERA as follows.

Theorem 1. Let Φ be an Alloy formula and let n be a given scope. The sequential circuit $C = \text{SERA}(\Phi, n)$ always terminates and on termination, the invariant evaluates to true iff Φ is satisfiable in scope n.

Proof. Theorem 1 follows from a straightforward induction on the number of nodes in the parse tree for Φ. We use the validity entries in Column 1 of Table 3.1 and the `depth` function in all components to establish a tight upper bound on the depth of C. The bound depends on the length of the formula and the scope. Note that for a given scope, the call `compVec[0].depth()` can be computed at compile time and thus is a constant.

Base case. In the base case Φ is either a `Sig` or an implicit relation. Both are trivial and always satisfiable since they are initialized with nondeterministic values.

We will provide proofs for the existential quantifier, Boolean negation, Boolean conjunction, set intersection, relational product, and transitive closure operators. The proof for the sets and relations prove the correctness of their membership and cardinality encodings. The proofs for the correctness of the remaining constructs, (other Boolean connec-

tives, universal quantifier, set union, set complement, etc.) are straightforward as they can be rewritten in terms of the above operators.

Case $\Phi = \neg F$ If Φ is satisfiable within the scope n then there exists a model for Φ that sets F to false. Using the induction hypothesis we deduce that there is a trace $\mathcal{Z}_F$ that sets the predicate output of SERA(F, n) to false and the validity output to true in k_F cycles. Since the predicate output of SERA(Φ, n) is equal to the negation of the output predicate of SERA(F, n) and the validity outputs are identical, then the trace $\mathcal{Z}_F$ sets SERA(Φ, n) to true. Conversely, if Φ is not satisfiable, then there is no model that sets F to false. Using the induction hypothesis, there is no trace that can set the predicate output of SERA(F, n) to false while the validity output is true. It follows that there is no trace that sets SERA(Φ, n) to true.

Case $\Phi = F \wedge G$ If Φ is satisfiable within the scope n, then there exists a model σ that satisfies both formulas F and G simultaneously. By the induction hypothesis, there exits a trace $\mathcal{Z}_F$ that sets the predicate and validity outputs of SERA(F, n) to true in k_F cycles and there is a trace $\mathcal{Z}_G$ that sets the predicate and validity outputs of SERA(G, n) to true in k_G cycles. Assume without loss of generality due to the commutativity property of $\wedge$ that $k_F < k_G$. Since σ satisfies both formulas F and G simultaneously, we conclude that the two traces $\mathcal{Z}_F$ and $\mathcal{Z}_G$ are consistent and do not assign conflicting values to components that are common between SERA(F, n) and SERA(G, n) at any cycle. Thus we can construct $\mathcal{Z}_\Phi$, a satisfying trace for SERA(Φ, n), by selecting assignments to common components from SERA(F, n) and SERA(G, n) from $\mathcal{Z}_F$ for the first k_F cycles, and assign the rest according to the trace $\mathcal{Z}_G$.

Conversely, if Φ was not satisfiable, then there is no model that can satisfy both F and G simultaneously. Assume there is a trace $\mathcal{Z}_\Phi$ that can set the predicate and the validity output of SERA(Φ, n) in k cycles. Examine the evaluation $\mathcal{Z}_\Phi$ assigns to the signatures and relations involved in SERA(F, n) and SERA(G, n) at the initial cycle. Using the induction hypothesis, these values must satisfy F and G and thus we have a contradiction.

Case $\Phi = \exists v \in \Gamma.F(v)$ If Φ is satisfiable within the scope n, then there is a model $\sigma \subseteq \text{range}(\Gamma)$ and there is an element $\alpha \in \sigma$ such that $F(v = \alpha)$ is satisfiable. Let C_Φ be the SERA component corresponding to the $\exists v$ statement and let C_Γ and C_F be the SERA components corresponding to Γ and F respectively. By the induction hypothesis, there exists a trace $\mathcal{Z}_\Gamma$ that sets C_Γ to a state matching σ in k_Γ steps, and there exists a trace $\mathcal{Z}_\alpha$ that sets the predicate and validity output functions of C_F to true in k_F steps after presenting C_F with α. The component C_Φ enumerates all the possible elements in $\text{range}(\Gamma)$, queries C_Γ for their membership and concurrently presents them to C_F for evaluation in case the membership test was valid. After at most $i \leq |\text{range}(\Gamma)|$ steps from the point it starts, it is guaranteed to find α. Since it accumulates a disjunction of the results of C_F, the first valid true return value of the predicate output function of C_F terminates the computation. Following the above steps we can construct a concatenated trace $\mathcal{Z}_\Phi$ of length $k_\Gamma + i \times k_F$ that sets the Boolean predicate function of C_Φ to true as well as its predicate validity function. Since $\text{range}(\Gamma)$ is bound to be either a set, or a relation with arity a, then $|\text{range}(\Gamma)| \leq n^a$ and $\mathcal{Z}_\Phi$ is finite.

Conversely, if Φ is not satisfiable, then C_Φ is guaranteed to try all models of $\text{range}(\Gamma)$ and complete execution in a finite number of steps (2^{n^a}). By the induction hypothesis, all elements in models that match C_Γ will not satisfy C_F. At the end of the iteration the output predicate of C_Φ will be set to false and its corresponding validity output will be asserted and the `while` loop will terminate. Consequently there exists no trace that satisfies the predicate output function of C_Φ.

Case $\Phi = A \cap B$ We prove the correctness of the membership predicate of $\text{SERA}(\Phi, n)$ and the correctness of the cardinality follows since it counts the correct checks of membership in $\text{SERA}(\Phi, n)$. The proof is similar to the proof for the conjunction case since the membership and validity output of $\text{SERA}(\Phi, n)$ is equal to the conjunction of the membership and validity outputs of $\text{SERA}(A, n)$ and $\text{SERA}(B, n)$ respectively.

Case $\Phi = A.B$ Checking $(v_1, v_2) \in \Phi$ is encoded as $\exists v.(v_1, v) \in A \wedge (v, v_2) \in$

B. The correctness of the membership check holds since we already proved the correctness for the existential quantification and conjunction cases. As for the cardinality check, the same argument holds as for the intersection case since the cardinality is encoded as a count of the true membership checks across the range of Φ.

Case $\Phi = *E$ We encode transitive closure using the iterative squaring technique. The proof that iterative squaring correctly computes transitive closure is given in [55].

□

3.8 Evaluation of SERA

We introduce our SERA implementation and we present the results of SERA in comparison to the Alloy Analyzer.

3.8.1 Implementation

Our implementation of SERA mirrors the description in Section 3.5. As illustrated in Figure 3.2, we parse the Alloy model into a DAG of signatures, relations, and operators with the root as the command to be executed and the leaves as the signatures and implicit relations. Note that we generate a DAG since our analysis tries to reuse syntax-equivalent nodes. The C++ subset we used to describe SERA along with the implied concurrency semantics and the bounds on integers guaranteed by scope finitization can be directly synthesized to sequential circuits described in VHDL similar to [38, 36].

We end up with a hierarchical VHDL design with an asserted signal designated as the invariant. We pass the VHDL to SixthSense which, in case of satisfiability, provides a trace that satisfies the invariant. Step 1 of Figure 3.3 illustrates the mapping of the trace to an Alloy instance. We map the initial values of the membership state of component `V` as two vertices and the initial values of the `bitMatrix` membership state of component `E` as the existence or absence of arcs with label `E` between the vertices.

3.8.2 Results

To evaluate SERA we chose three examples when we began this research. The tree integrity entries in Table 3.5 show results for checking the `EquivOfTreeDefns` assertion. The other two examples are representatives from the standard Alloy distribution that have been the subject of research by multiple Alloy related papers. The file system example describes relations between directories, files, and a root directory in a Unix-like file system and asserts alias consistency and acyclicity. The LISP list example describes empty and non-empty nested lists of objects, defines equivalency between lists, and asserts symmetry and reflexiveness properties of the equivalence definition; it also asserts that all empty lists are equivalent.

For the unsatisfiable formula from the tree example, the Alloy Analyzer could not perform checks with a scope larger than 6; SixthSense applied to SERA-generated sequential circuits was able to automatically check these formulas for scopes upto 32. The ∞ in the Alloy entries denotes a timeout with a time limit of $14,000$ seconds. For unsatisfiable formulas from the list example, the Alloy Analyzer failed beyond a scope of 9, whereas SERA could check these formulas for scopes upto 32. For satisfiable formulas from the list and file suites, we specified lower bounds on the minimum size of the list and file examples as Alloy `facts`, and were able to find counter-examples in scopes $3\times$ larger than the Alloy Analyzer.

In the size columns of Table 3.5, $|V|$ and $|C|$ denote the number of CNF variables and clauses respectively, and $|\mathcal{I}|$, $|\mathcal{R}|$, and |ANDs| denote the number of sequential circuit primary inputs, registers, and AND gates respectively as a measure of their complexity [73]. The time column indicate the time it took the solver to decide the CNF formula or the sequential circuit for a given scope.

We indicate whether a formula is satisfiable or not in the title of each sub-table. We ran all experiments on a 1.7 GHz Pentium 4 machine with 1 GB memory. For our examples, the Berkmin solver consistently outperformed all the other solvers that come

with the standard Alloy distribution, so we tabulate the results for Alloy using Berkmin. The Alloy Analyzer was able to validate the tree equivalence for scopes up to 6. For a scope of 7, the SAT solvers spaced out (∞), and for a scope of 8 the Alloy Analyzer ran out of memory and could not generate the CNF formula.

In general, we noticed that the number of needed memory elements grew quadratically with the scope and this agrees with the highest complexity of SERA. Using sequential encoding, we were able to scale Alloy analysis to a scope of 32 with relatively acceptable computational resources and time limitations.

All cases required applying iterative reduction transformations. In the case of the tree equivalence example, localization abstractions were instrumental in reducing the problem, also equivalence detection did a good job of merging the common parts of the different equivalent tree definitions. In the file system case, the counter-examples happened to be relatively sequentially deep since they depended on comparisons between transitive closures, high cardinality comparisons, and conflicting transpose statements. Semi-formal search was able to detect counter-examples once the design was reduced using equivalence merging.

3.9 Summary

We developed the use of sequential circuits for checking the validity of Alloy formulas. By doing so we used far fewer variables and enabled sophisticated automatic reduction techniques to be applied. We were able to show that a scope of 32 is feasible using reasonable resources.

Table 3.4: Pseudo C++ description for universal quantifier and transitive closure components.

```
template<class Sig>
class ForAll:public Component{
  Sig & V;
  Sig::iter & v;
  Component & F;
  bool value, valid;

  bool predicate() { return value;};
  bool predValid() { return valid;};

  void initialState(){
    value = true;
    valid = false;
    v = 0;};
  void nextState(){
    if (!valid && v.isValid()){
      if (F.predValid()){
        value &= F.predicate();
        if (v.isLast() ){
          valid == true; }
        if (!value){
          valid = true;
          F.terminate();}
        v++;
        F.evaluate();
      } } };

  int depth(){
    return Sig::scope*F.depth();
  };};
```

(a) Universal quantifier component

```
template<class S1>
class TClosure:public Relation{
  Relation<S1,S1> & T;
  bitMatrix E;
  bool valid;
  int count;

  bool in(S1::iter u,S1::iter v){
    return E[u][v];};
  int card(){
    return countOnes(E);};

  bool memberValid() { return valid;};
  Boolean cardValid() { return valid;};

  void initialState(){
    E = T.R;
    count = 0;
    valid = false;};
  void nextState(){
    if ((!valid) && T.cardValid() &&
        T.V1.cardValid() &&
        T.V2.cardValid()){
      E = E*E + E;//iterative squaring
      int k = max( lg(T.V1.card()),
             lg(T.V2.card()));

      if (count++ == k)
          valid = true; } };

  int depth(){
    int n=max(S1::scope, S2::scope);
    return lg(n)+T.depth();};};
```

(b) Transitive closure component

Table 3.5: Results of SERA and Alloy Analyzer.

Tree integrity — unreachable					
Technique	Scope	Size			Time (sec)
Alloy Analyzer	5	$\|V\|$=212,032	$\|C\|$=641,644		156
	6	$\|V\|$=641,983	$\|C\|$=2,014,005		658
	7	$\|V\|$=1,682,479	$\|C\|$=5,428,222		∞
	8	timed out before generating CNF			∞
SERA	4	$\|\mathcal{I}\|$=12	$\|\mathcal{R}\|$=36	$\|$ANDs$\|$=354	13
	8	$\|\mathcal{I}\|$=32	$\|\mathcal{R}\|$=94	$\|$ANDs$\|$=1,087	18
	16	$\|\mathcal{I}\|$=80	$\|\mathcal{R}\|$=780	$\|$ANDs$\|$=10,786	220
	32	$\|\mathcal{I}\|$=192	$\|\mathcal{R}\|$=3,272	$\|$ANDs$\|$=21,841	4,309

File System — reachable					
Technique	Scope	Size			Time (sec)
Alloy Analyzer	5	$\|V\|$=4,703	$\|C\|$=14,448		5
	6	$\|V\|$=7,204	$\|C\|$=22,640		6
	7	$\|V\|$=10,377	$\|C\|$=33,232		21
	8	$\|V\|$=13,448	$\|C\|$=43,875		28
SERA	4	$\|\mathcal{I}\|$=22	$\|\mathcal{R}\|$=112	$\|$ANDs$\|$=1,054	26
	8	$\|\mathcal{I}\|$=41	$\|\mathcal{R}\|$=728	$\|$ANDs$\|$=4,548	39
	16	$\|\mathcal{I}\|$=68	$\|\mathcal{R}\|$=4,264	$\|$ANDs$\|$=19,545	417
	32	$\|\mathcal{I}\|$=143	$\|\mathcal{R}\|$=27,720	$\|$ANDs$\|$=421,681	1,712

Lisp Lists — reachable					
Technique	Scope	Size			Time (sec)
Alloy	10	$\|V\|$=17,035	$\|C\|$=98,381		36
SERA	32	$\|\mathcal{I}\|$=178	$\|\mathcal{R}\|$=22,305	$\|$ANDs$\|$=285,046	171

Lisp Lists — unreachable					
Technique	Scope	Size			Time (sec)
Alloy Analyzer	8	$\|V\|$=2,990	$\|C\|$=68,858		55
	9	$\|V\|$=18,921	$\|C\|$=103,440		2,062
	10	$\|V\|$=23,704	$\|C\|$=129,628		∞
SERA	8	$\|\mathcal{I}\|$=46	$\|\mathcal{R}\|$=1,020	$\|$ANDs$\|$=13,859	229
	16	$\|\mathcal{I}\|$=88	$\|\mathcal{R}\|$=5,226	$\|$ANDs$\|$=110,734	341
	32	$\|\mathcal{I}\|$=178	$\|\mathcal{R}\|$=25,790	$\|$ANDs$\|$=328,065	575

Chapter 4

Sequential Encoding for Imperative Programs

4.1 Bounded model checking for ANSI-C programs

Several static analysis techniques have emerged lately addressing the verification of software programs [25, 74, 75, 76]. Software programs are undecidable and thus static analysis tools often resort to abstraction and finitization techniques to render them amenable to model checking [2].

CBMC [25] is a bounded model checker for ANSI-C programs that checks for properties such as pointer safety and within-bound array access as well as user assert statements. Given an ANSI-C program and a bound on the range of variables therein, CBMC computes a Boolean formula that asserts the desired properties in the program. It does that by unwinding a complex state machine that describes the program and its properties into a CNF Boolean formula and checks the formula using a SAT procedure [27, 31, 32] to look for a counter-example.

SAT solvers often face an exponential blow up in the number of possible assignments to the atomic propositions. This problem along with the large number of variables

used in the CNF encoding, often limits the SAT-based CBMC analysis to restricted variable ranges. By scaling the analysis of ANSI-C programs to larger bounds, we increase its applicability to real-world designs.

4.1.1 Sequential circuits for program analysis

While recent advances in SAT have enabled checking properties of designs of real systems, these implementations often need to be substantially incomplete, leaving out important aspects of the systems, to enable the analysis to complete. Moreover, the analysis is typically limited to relatively small bounds, e.g., fewer than 16 entries in an array sort program as we will demonstrate in Section 4.5.

To extend the applicability of static analysis to a wider class of programs as well as to check more sophisticated properties and gain more confidence in the results, we need to scale the analysis to significantly larger bounds.

The limitations of the CNF encoding, as discussed in Section 1.4, motivated us to develop sequential encoding for bounded ANSI-C program analysis (SEBAC), an algorithm which encodes ANSI-C programs as sequential circuits and decides them using a sequential circuit solver. A sequential circuit can be viewed as a restricted C++ program, specifically a multi-threaded program in which all variables are either Boolean-valued or integers, whose range is statically bounded, and unbounded allocation is forbidden [50, 36].

Given an ANSI-C program and a bound, SEBAC automatically derives a sequential circuit and a Boolean variable therein that serves as an invariant, i.e., the variable can be set to true if and only if a property that the program asserts is violated within the bound.

In Chapter 3 we demonstrated that sequential circuit analysis scaled to bounds orders of magnitude higher than SAT analysis for checking satisfaction of formulas expressed in declarative relational first order logic with transitive closure [50].

In this chapter we study the benefits for the analysis of imperative programs. We note that there are two key advantages to compiling ANSI-C programs into sequential cir-

Table 4.1: CBMC transformation of ANSI-C programs into a Boolean formula.

(a) simplify

```
x = x + y;
if (x != 1) {
  x = 2;
  if (z)
    x++;
}

assert (x <= 3);
```

(b) rename

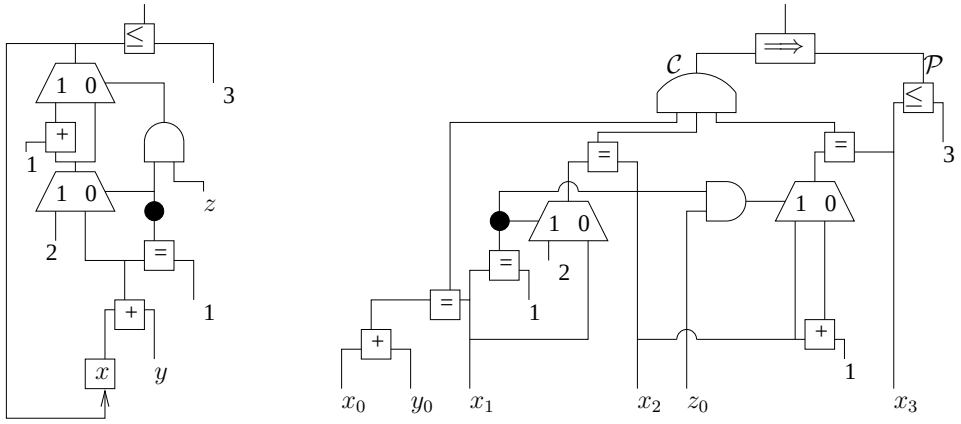
```
x1 = x0 + y0
if (x1 != 1) {
  x2 = 2;
  if (z0)
    x3 = x2 + 1;
}

assert (x3 <= 3);
```

(c) formula

$\mathcal{C}$:= $(x_1 = x_0 + y_0) \wedge$
$(x_2 = ((x_1 \neq 1)\ ?\ 2 : x_1)) \wedge$
$(x_3 = ((x_1 \neq 1 \wedge z_0)\ ?\ x_2 + 1 : x_2))$

$\mathcal{P}$:= $x_3 \leq 3$
$\mathcal{C} \wedge \neg\ \mathcal{P}$



(a) Sequential circuit (b) Boolean formula before CNF

Figure 4.1: Sequential circuit encoding versus the CBMC Boolean formula.

cuits rather than CNF formulas:

Advantage 1 Our encodings are much more succinct than those generated by CBMC—in cases, CBMC encoding algorithms produce a data structure that uses several orders of magnitude more memory to represent. For instance, CBMC needed a CNF formula with 4.7 million variables and 18.9 million clauses to encode an array sorting routine with a bound of 16 on the size of the array and the range of the values for the array entries.

Advantage 2 Casting the decision problem for a property of an ANSI-C program as an

invariant check on a sequential circuit allows us to make use of a number of powerful automated analysis techniques that we discussed in Section 2.4 and that have no counterpart in CNF analysis. Empirically, our results show that SEBAC scales to bounds that are orders of magnitude higher compared to CBMC.

We make the following key contributions:

1. **Sequential analysis:** We enable the use of sequential circuit verification including many powerful reduction techniques for ANSI-C model checking.

2. **Create verification flow for existing encoding:** We enable a software static analysis flow from SPARK, a fully automated hardware high-level synthesis tool that translates C to sequential circuits [51]. SPARK has not been used for verification purposes before.

3. **New encoding for ANSI-C programs:** We propose SEBAC, a novel algorithm to encode an ANSI-C program with a bound on ranges of variables into a sequential circuit. The SEBAC encoding is more optimal for verification purposes than the SPARK encoding as the latter targets optimizations such as time multiplexing and circuit area reduction at the expense of increasing the number of variables in the circuit.

4. **Evaluate TBV for C programs:** We evaluate TBV analysis of C programs by comparing SPARK and SEBAC coupled with a TBV solver against CBMC coupled with SAT. We apply these techniques to find real and subtle code bugs that were reported by Adam Barr from Microsoft as challenging and require careful code inspection to be revealed [77]. As our results show, sequential analysis techniques scale to bounds orders of magnitude higher than CNF and SAT techniques.

4.2 Illustrative example

We show in Column (a) of Table 4.1 the same example code used to illustrate the CBMC translation from ANSI-C into CNF in [25]. The first step is to transform the code into a static single assignment form where each variable is assigned only once. This is done by introducing new variables through variable renaming and the result is shown in Column (b) of Table 4.1. Then CBMC computes a set of constraints $\mathcal{C}$ and properties $\mathcal{P}$ and builds a Boolean formula $\mathcal{C} \wedge \neg\mathcal{P}$ by treating variables as bit vectors. A satisfying valuation to variables of this formula represents a counter-example. We show the combinational circuit corresponding to $\mathcal{C} \wedge \neg\mathcal{P}$ in Figure 4.1(b). The formula is then flattened into CNF which has only two levels of logical hierarchy ($\bigwedge_{i=1}^{n} \bigvee_{j=1}^{m_i} l_{ij}$) where n is the number of clauses, m_i is the number of variables in clause i, and l_{ij} is a literal expressing either the variable indexed by j in clause i or its negation). In the process, the structure of the circuit is lost. In contrast, the sequential circuit encoding shown in Figure 4.1(a) introduces no new variables and keeps the hierarchy of the program.

Note that in case the code in Column (a) of Table 4.1 was enclosed in a loop, CBMC would assume an unwinding bound W and eventually replicate the same circuit in Figure 4.1(b) W times with new variables each time. If the check on the CNF formula resulting from the unwinding with W fails because the unwinding was not enough, CBMC would increase W and re-encode the CNF formula. In contrast, the sequential circuit encoding in Figure 4.1(a) stays intact and expresses the loop via updating the state of x through executing the circuit as much steps as needed per the bounded model checker.

4.3 CBMC, sequential circuit analysis, and SPARK

In this section we introduce the CBMC analysis of ANSI-C programs. We briefly review sequential circuits. We compare our encoding in SEBAC, which targets optimizations useful for static analysis, to the output of SPARK [51]; a fully automated high-level synthesis

tool that generates sequential circuits out of restricted C programs. SEBAC performs better since it infers the sequential structures from within the program while SPARK treats every statement as a sequential step and uses various techniques to find statements that can be executed concurrently. Finally, we describe how transformation-based verification checks properties of sequential circuits.

4.3.1 CBMC

CBMC reduces the check of a property of an ANSI-C program with bound N on the range of variables and with bound W on the number of loop and recursion unwinding to the problem of determining the satisfiability of a Boolean formula.

To translate an ANSI-C program, CBMC unwinds loop constructs that can occur as `for` and `while` loops, `goto` statements, and recursive function calls. The `for` and `while` loops are unwound W times where W is the unwinding bound. An assertion is added to the last copy to ensure that the loop does not need more iterations to complete the computation. An unwinding assertion guides CBMC to increase W for a certain loop in case the unwinding assertion fails. Recursive function calls are assumed to recurse to a bounded depth and an assertion similar to the loop unwinding assertion is added to ensure that depth is enough. The `goto` statements inducing loops are handled similarly.

Function calls are inlined within the calling function. The `return` statements are replaced by assignments and `goto` statements pointing to the end of the function expansion. This construction results into a simplified program with `if`, forward `goto`, assignment, assertion, and arithmetic statements as well as labels defining branching targets.

Table 4.1 illustrates through an example the translation from the simplified ANSI-C program into a static single assignment form. Column (a) in Table 4.1 shows a simplified piece of code. CBMC renames the variables so that each variable is assigned only once and the result is shown in Column (b) of Table 4.1. In the process variables x_1, x_2 and x_3 were added. Then CBMC computes a set of constraints $\mathcal{C}$ on all the variables, including the

ones it introduced, and a set of properties $\mathcal{P}$ deduced from the assertions in the simplified program. CBMC translates the formula $\mathcal{C} \wedge \neg\mathcal{P}$ into a CNF formula by considering each variable as a bit vector with sufficient width to represent the bound on the variable range.

When CBMC introduces new variables in the so-called renaming transformation, it is actually embedding internal hierarchical nodes in the structure of the program as illustrated in Figure 4.1(b). However, CBMC loses this hierarchy when it translates $\mathcal{C} \wedge \neg\mathcal{P}$ into CNF because CNF is a flat Boolean formula with only two levels of hierarchy (AND of ORs). Also CBMC needs to introduce new variables to represent the internal nodes since its final target (CNF) is stateless.

4.3.2 Sequential circuits

In Figure 4.1(a) we show a sequential circuit encoding of the ANSI-C program from Table 4.1. It keeps the exact hierarchy of the ANSI-C program and does not require additional variables since it has registers; memory elements that keep the state of the circuit.

The sequential circuit in Figure 4.1(a) has one register x and two inputs y and z. The initial state of x sets x to a nondeterministic value, and the next state is a function of x, y, and z. The sequential circuit needs one step to complete and the output function returns the assertion in Column (a) of Table 4.1.

We compare that to Figure 4.1(b) which shows the Boolean formula CBMC generates before translating it to CNF. Notice that this formula has yet to be flattened into CNF and thus more variables may be introduced to transform it into two levels of hierarchy only.

4.3.3 C to sequential circuits using SPARK

SPARK is a fully automated high-level synthesis tool [51] that is designed for implementing systems and not for verifying them. It takes a subset of ANSI-C constructs and bounds on variable ranges as input and produces a synthesizable register-transfer level VHDL that describes a sequential circuit as output. Briefly, SPARK recognizes atomic statements in

the ANSI-C code and constructs a state machine that executes a statement per step. Then SPARK uses high-level synthesis techniques to schedule the execution of these statements in order to optimize performance, reduce the area of the circuit, or better utilize the limited hardware resources [51, 78]. SPARK performs renaming techniques and a set of heuristic transformations called code motions to move and merge the execution of the atomic statements inside and outside of conditionals and loop constructs. By doing so SPARK introduces new sequential elements to the ANSI-C program by assuming the execution of one statement per step. Then it tries to reduce the number of steps it needs to execute the ANSI-C program via heuristics that allow merging of these steps.

In Figures 4.2 and 4.3 we show how SPARK translates the example in Table 4.1 into a sequential circuit. Figure 4.2 shows the control flow circuit that implements the schedule computed by SPARK. Figure 4.3 shows the data flow circuit that executes the atomic computation statements. One can think of the two figures as two processes that execute concurrently and share variables. The variable `state` transitions between states $S_0, \ldots, S_5$, and done. SPARK introduced one bit-vector valued variable that encodes the state of the circuit and controls the schedule of execution. This is substantially less than the variables CBMC introduces.

The optimizations and transformations SPARK performs are not targeted to obtain a sequential circuit that is better amenable for static analysis. SPARK targets optimizations like choosing between complex structures such as ripple carry versus tree adders, reducing gate delays, and time multiplexing operations onto a few functional units [79] [80, Chapter 6].For example, we see that in Figure 4.3 SPARK minimized the logical depth of all combinational functions on the expense of needing more steps to complete the computation. We also observe that SPARK introduces additional sequential behavior to the program through its separate schedule state machine in order to provide a flexible infrastructure for the so called code motion transformations. For example, and without going into details, some code motions can be done by simply assigning new encodings for the states S_0 through S_5

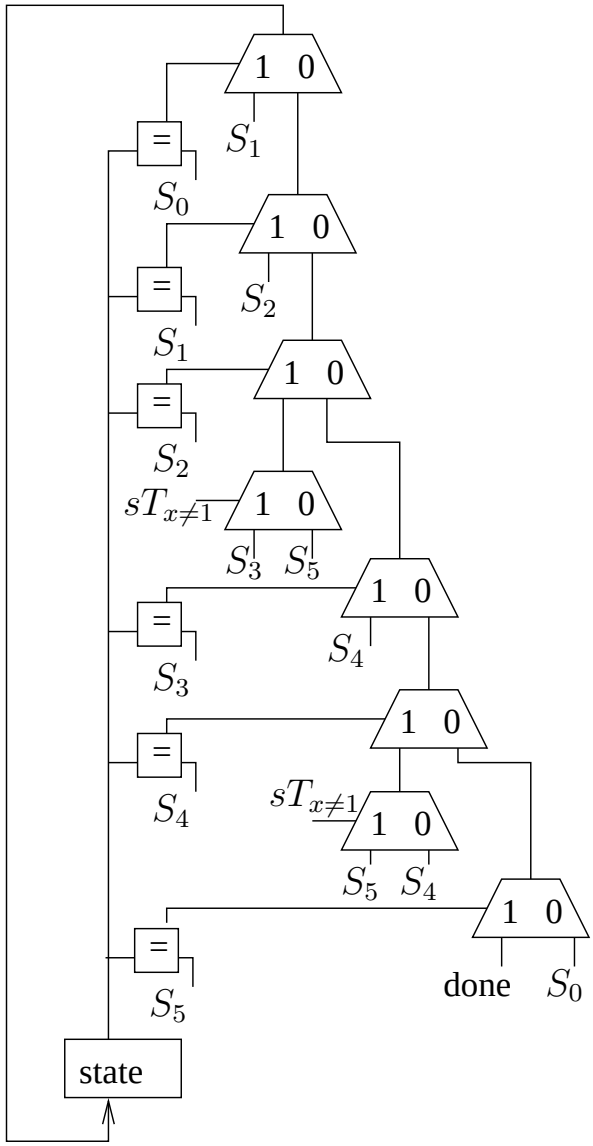


Figure 4.2: Control flow circuit.

to allow the execution of more than one statement at the same time.

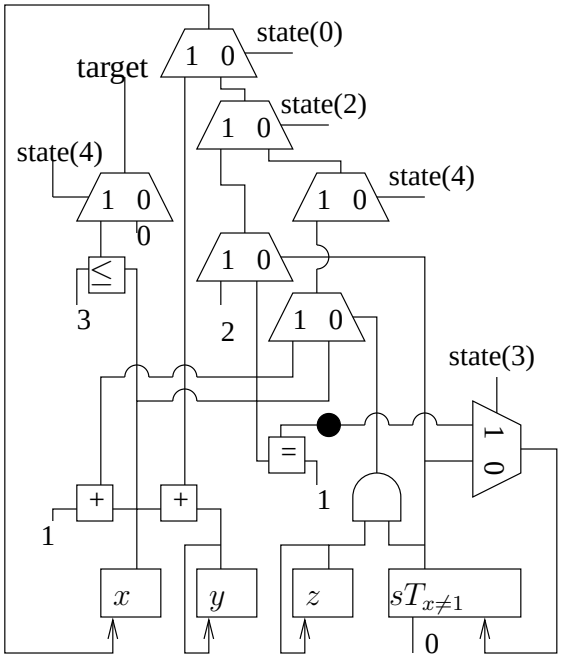


Figure 4.3: Data flow circuit.

4.4 SEBAC

Given an ANSI-C program with assertions therein and a bound on the ranges of variables, SEBAC produces a sequential circuit and a Boolean invariant therein such that there is a trace of the sequential circuit that sets the invariant to true if and only if an assertion is violated.

Recall that CBMC translates an ANSI-C program with a bound on the ranges of variables to a Boolean formula via (1.) introducing new variables and (2.) embedding a constraint system hierarchy into the program. Then CBMC loses the structure it introduced when it flattens the Boolean formula into CNF before passing it to SAT. SPARK, in contrast, uses fewer variables than CBMC to encode the bounded program into a sequential circuit, it actually introduces state and sequential elements to the program that are not necessary for its execution. It also adds scheduling complexity and encodes the schedule control states in

a way that allows flexible code motions on the expense of adding more Boolean variables.

4.4.1 Overview

Intuitively, SEBAC aims at reducing the number of variables needed to sequentially encode the ANSI-C program and at reducing the sequential depth needed for the circuit to complete execution, thus allowing bounded model checking a better chance to complete the analysis. Unlike CBMC that uses the assignment statements to build a constraint hierarchy that gets lost later after translation to CNF, SEBAC uses the data structures and the branching logic in the program to build a circuit hierarchy.

In contrast to SPARK, SEBAC infers sequential elements from the semantics of the program rather than by introducing sequential elements based on the ANSI-C semantics of sequential statements. SEBAC makes use of sequential elements such as dependent code blocks and loop iterators. To achieve this, SEBAC simply ignores high-level synthesis techniques, and translates the ANSI-C program into literally syntactically equivalent VHDL processes that use high-level VHDL constructs such as records, loops, and overloaded functions to match the ANSI-C high-level constructs. It also makes use of VHDL generic parameters to simplify specifying bounds for the program.

4.4.2 Correctness of SEBAC

Similar to Clarke *et al.* [81], we follow the C99 semantics of ANSI-C programs as modeled in [82]. The formal description and proof of correctness of SEBAC is based on specifying a formalization for C and sequential circuits, and a recursive algorithm that takes nodes from the parse tree and the data flow diagram of the C program P and maps them to a sequential circuit SEBAC(P) with an invariant $A(P)$. The invariant $A(P)$ is satisfied if and only if an assertion is violated in P.

We will demonstrate the equivalence of the SEBAC encoding to that of CBMC for those constructs where the encoding differs from or does not match directly with the

ANSI-C semantics as in Boolean and integer arithmetic and assignments. As discussed in Section 4.3.1, CBMC assumes the program is already preprocessed, and performs a series of transformations and rewrites to the program. The first transformations expand function calls and replace `break`, `continue`, `switch/case`, `for`, and `do/while` statements with equivalent `if/else` and `while` statements.

CBMC treats recursive functions and loops inferred from backward `goto` statements the same way it treats `while` loops—it unwinds them. These transformations leave us with an ANSI-C program that consists of `if` statements, assignments, assertions, and `while` loops.

We limit our attention to the correctness of encoding loops and pointers since the rest of the C constructs map directly to their counterparts in VHDL, describing sequential circuits. These constructs have the same semantics as the HOL [83] formalization for C [84] and VHDL [85] suggest. The HOL system consists of a functional programming language with strong polymorphic typing called Meta-Language (ML). Terms and theorems are ML datatypes and inference rules are ML functions.

CBMC performs additional transformations such as variable renaming and loop unwinding. A one step unwinding of `while(g) s` will be `if(g)`$\{$`s; while(g) s`$\}$. After a number of unwinding steps equal to the unwinding bound, the tailing `while(g) s` loop is replaced with an assertion `assert(!g)`. The assertion is essential to guarantee that the unwinding was enough for the loop to complete its computation. If the assertion fails for any possible execution, then the number of unwindings is increased until it is large enough.

SEBAC builds a sequential component for the same loop that executes `s` and evaluates `g` and sets the `done` property only when `g` evaluates to false. The semantics are similar to `while(!(done = !g)) nextState(l);` with `void nextState(l)`$\{$ s_l`;`$\}$ where each iteration is a cycle as described in Section 2.3.3. The parameter list `l` accepts addresses to the variables that are used in s but are not declared in s, and $s_l = s|_{d:(*d),d\in l}$

substitutes each use of d in s by a dereference of d. We justify the equivalence of s and s_l by the equivalence of the expressions $(*\&d)$ and d. Intuitively, The encoding is equivalent to the original loop with a simple expansion of the body of `nextState` and a substitution of `done` with the right handside of the assignment. We prove the equivalence of the encoding to the original loop by following the HOL [83] formalization of the C language in [84].

Theorem 2. The statements `while(g) s` and `while(!(done=!g)) s` are equivalent under the HOL formalization of the C language in terms of the number of iterations they execute the statement `s`.

Proof. We make use of the operational semantics rules from [84] governing variable evaluation, the assignment operator, and the negation operator. An identifier d under a state of the program σ is evaluated as follows.

$$\frac{}{\langle d, \sigma\rangle \mapsto_e \langle \mathrm{LV}(A_\sigma(d), \Gamma_\sigma(d)), \sigma\rangle}$$

Here, $\mapsto_e$ denotes an expression evaluation relation whose action can be seen as a gradual transformation of a piece of syntax into a value. A value is a pair (m, τ) where m is a sequence of bytes and τ is the type of the value. $\mathrm{LV}(m, \tau)$ returns the left value of concrete value m of type τ, $A_\sigma(d)$ returns the address of identifier d under state σ, and $\Gamma_\sigma(d)$ returns the type of identifier d under state σ.

An expression $!e$ where e is an expression evaluating to a scalar type τ is evaluated as follows.

$$\frac{}{\langle !(m, \tau), \sigma\rangle \mapsto_e \langle (m \neq 0, \tau), \sigma(d))\rangle}$$

The expression $m \neq 0$ evaluates to a non-zero value when m is different than zero and to a value of zero otherwise.

An assignment operation $e_1 = e_2$ where e_1 and e_2 is reducible as follows.

$$\frac{\langle e_2, \sigma_0\rangle \mapsto_e \langle e_2', \sigma\rangle}{\langle e_1 = e_2, \sigma_0\rangle \mapsto_e \langle e_1 = e_2', \sigma\rangle}$$

$$\frac{\langle(\tau)(m_0,\tau_0),\sigma_0\rangle \mapsto_e \langle v,\sigma_0\rangle}{\langle LV(a,\tau)=(m_0,\tau_0),\sigma_0\rangle \mapsto_e \langle v,\sigma\rangle}$$

These are two simplified rules compared to the original rules in [84] as the original rules treat also compound assignments such as `+=`.

The `while` loop statement is defined as follows.

$$\text{while}(g)s \ \triangle= \ \mathcal{T}(\text{breakVal}, \mathcal{O}(g, \mathcal{T}(\text{contVal}, s)))$$

Here, the construct $\mathcal{T}$ is a trap structure that monitors the value of the statement during its execution. If the loop executes a `continue` statement, $\mathcal{T}$ captures a contVal value, stops the execution of the rest of `s` and iterates. In case the loop executes a break statement, $\mathcal{T}$ captures a breakVal value and forces the loop to stop.

The loop construct $\mathcal{O}$ takes a guard g and a statement `s`. In case the guard expression g evaluates to zero, $\mathcal{O}$ terminates, otherwise $\mathcal{O}$ iterates. In cases the expression may have an undefined value (due to a division by zero for example) and in that, the behavior of $\mathcal{O}$ is not defined.

Assume the guard g evaluates to m_0 under the state σ_0 and applying the rules for negation, assignment, variable, and negation evaluation in sequence to evaluate the `!(done=!g)` expression we obtain $(((m_0 \neq 0) \neq 0) \neq 0)$ under the state $\sigma_0 \cap \text{done} = ((m_0 \neq 0) \neq 0)$. We know that m_0 and $(((m_0 \neq 0) \neq 0) \neq 0)$ evaluate to the same value by simply assuming m_0 is equal to zero, different that zero, or undefined. Since the variable `done` is added to the program and thus unused in `s`, we conclude that the two loops `while(g) s` and `while(!(done=!g)) s` execute the statement `s` the same number of times. □

Theorem 3. Given the definition `void nextState(l)`$\{s_l\}$, the statements `s` and `nextState(l);` are equivalent under the HOL formalization of the C language.

Proof. The rule to evaluate a function $\tau_c f(\text{args})c$ is given as follows.

$$\frac{\langle c, \text{instParms}(\sigma_0, \text{args}, [e_1, \ldots, e_n])\rangle \mapsto_c \langle v_c, \sigma\rangle}{\langle \widehat{(m, \tau)}(e_1, \ldots, e_n), \sigma_0\rangle \mapsto_e \langle v, \sigma_0[M := M\sigma, I := I\sigma \cap \Lambda_{\sigma_0}]\rangle}$$

The function `instParms` is defined over the state σ_0 of the program, a list of arguments `args`, and a list of evaluations to the arguments $e_1, \ldots, e_n$. The relation $\mapsto_c$ is the relation governing the execution of the body of the function expressed in the statement c. The lower part of the rule states that the function call alters the state by only substituting the memory alterations M_σ due to executing c into the memory state M, and the allocated addresses Λ_σ due to the execution of c into the initialized addresses I.

The function `instParms` is defined inductively over the list of declarations of its parameters as follows.

$$\text{instParms}(\sigma, (d, \tau) :: \text{tail}, (v_0, \tau_0) :: vs) =$$

$$\text{instParms} = (\sigma \left[\begin{array}{lcl} A & := & A^g[d := a], \\ \Gamma & := & \Gamma^g[d := \tau], \\ I & := & I \cup r, \\ \Lambda & := & \Lambda \cup r, \\ M & := & M[\langle a \ldots\rangle := (\tau_0 \mapsto \tau)(v_0)], \\ \Sigma & := & \Sigma^g \end{array}\right], \text{tail}, vs)$$

A pair (d, τ) denotes the parameter identifier d and its type τ. A pair (v_0, τ_0) denotes the value corresponding to (d, τ) passed in an `instParms` function call. The rest of the parameters and their values are denoted with `tail` and `vs` respectively. The base case where the parameter list is empty is given by $\text{instParms}(\sigma, \text{empty}, \text{empty}) = \sigma$.

The components of the program state σ are A, Γ, I, Λ, M, and Σ. They respectively denote the variable addresses, the type environment, the initialized addresses, the allocated addresses, the memory map, and the text of the program. The function call adds an address into A for every parameter d and initializes the memory corresponding to d in M starting at

address a with the cast of v_0 from τ_0 to τ where τ_0 is the type of the value passed and τ is the declared type in the parameter list. The allocated addresses Λ and the initialized addresses I sets also get updated with the corresponding sequence of bytes $r = \{a \ldots a + |\tau| - 1\}$ of size $|\tau| - 1$ starting at a. The type environment Γ gets updated with the relation between d and τ. The superscript ^{g} denotes a global set.

The arguments passed to `nextState` are all addresses of exact pointer types and we are guaranteed that they are only accessed after dereferencing. Thus, A, Γ, I and Λ are not changed. Following the function call evaluation rule and the definition of `instParms` we notice that the only change to the state of the program that is induced by the call to `nextState` is that induced by its body expressed as s_l to the memory of the program expressed in M. The changes to M by s and s_1 in the context of `nextState` are the same since $(*\&d)$ is equivalent to d. Thus `while(g) s` is equivalent to `while(g) nextState(l);`. □

Combining the results of Theorems 2 and 3, we conclude that `while(!(done=!g)) nextState();` is equivalent to `while(g) s`.

SEBAC treats pointers as indices into an array representing memory. This is correct since in ANSI-C `*p` is equivalent to `p[0]` and `*p +i` is equivalent to `p[i]` where `p` is a pointer to memory and `i` is an integer.

4.4.3 Mapping C to VHDL

We illustrate the process of mapping C to VHDL by looking at a buggy array selection sort algorithm from [77] with correctness properties checking whether the array is in order and whether an arbitrary entry in the array still exists in it after the sort. We highlight interesting constructs and explain the details of the sort while illustrating the mapping.

Selection sort with bug in ANSI-C

```
1  /*! \brief array selection sort routine */
2  void selsort (int a[], int size) {
```

```
3    int current, j, lowestindex, temp;
4
5    for (current = 0; current < size - 1; current++) {
6      lowestindex = current;
7      /* find the index of the lowest value */
8      for (j  = current + 1; j < size; j++) {
9        if (a[j] < a[current]) {
10          lowestindex = j;
11       }
12     }
13     if (lowestindex != current) {
14       /* swap a[current] and a[lowestindex]
15        * since difference exists */
16       temp = a[current];
17       a[current] = a[lowestindex];
18       a[lowestindex] = temp;
19     } } }
20
21  /*! \brief checker for array selection sort */
22  void selsortproperty (int size) {
23    int a[MAXSIZE];
24    int i, iTest, jTest, aTest;
25
26    if (size == 0) return;
27    iTest = iTest % size;
28    jTest = jTest % size;
29    aTest = a[iTest];
30
31    selsort (a, size);
32
33    if (iTest < jTest )
34      assert ((a[iTest] <= a[jTest]));
35
36    for (i = 0; i < size; i++ )
```

```
37      if (aTest == a[i])
38        break;
39    assert(aTest == a[i]);
40  }
```

C Function to VHDL entity. SEBAC translates the `selsortproperty` function into a VHDL entity. The entity takes `size` as an input and `bound` as a generic parameter that configures the range of `integer` variables.

```
entity selsortproperty is
  generic (bound : integer := 64);
  port (signal size : integer range 0 to bound - 1);
end;
```

The behavior of the entity is described in a corresponding VHDL `architecture`.

```
architecture selsortproperty of selsortproperty is
  constant bound_lg2: integer := util_log2(bound);
  signal a : IntArray( 0 to bound - 1);
  signal i, iTest, jTest, aTest : integer range 0 to bound - 1;

  signal current: integer range 0 to bound - 1;
  signal currentV, currentVN : std_ulogic_vector(0 to bound_lg2);
```

Sequential loops. The header section in the architecture declares signals that represent `selsortproperty` and `selsort` variables as SEBAC inlines function calls. Signal `a` is declared as an `IntArray`, which is an array of integers defined in a custom package. Signals `currentV` and `currentVN` are defined as logic vectors.

```
begin
  currentVN <= currentV when currentV = size else currentV + 1;
  currentV <= util_latch(currentVN);
  current <= util_type_conv(currentV);
```

Signal `currentVN` is the next state function of `currentV` and is connected to it through a latch function. The signal `current` is the only sequential iterator in the selection sort algorithm and plays the role of a program counter that schedules the execution of the algorithm. It is connected to `currentV` with a type conversion function which means that it is only a *wire* and no additional variables are needed for it.

```
p1: process(current)
  variable lowestindex, j: integer range 0 to bound - 1;
begin
if (size /= 0 ) then
  if (current = 0 ) then
    aTest <= a(iTest);
  end if;
```

Function inlining. The VHDL process `p1` defines the variables needed in the inner loop of the selection sort algorithm. Note that `lowestindex` is declared as a variable and not as a signal since it does not carry information or state and is only used within process `p1` to compute a value. The check on Line 26 of `selsortproperty`, that exits the function with a `return` statement, is translated to the `if (size /= 0)` condition that encloses the process body. SEBAC inlines statements from both functions and thus stores in `aTest` the value of an arbitrary entry from the array indexed by `iTest` which is left uninitialized to denote a nondeterministic index.

Loops with dynamic bounds. SEBAC encloses the body of the loop from Line 5 in a conditional statement. The lowest index is computed at each execution step through the translation of the loop on Line 8.

```
if (current < size - 1) then
  lowestindex := current;
  for j in 0 to bound - 1 loop
    if (j < current + 1) then
      next;
    end if;
```

```
      if (j < size) then
        if (a(j) < a(lowestindex)) then
          lowestindex := j;
        end if;
      end if;
    end loop;
    if (lowestindex /= current) then
      a(lowestindex) <= a(current);
      a(current) <= a(lowestindex);
    end if;
  end if;
```

Since SixthSense does not allow loops with dynamic bounds, we encode the ANSI-C `for` statement into a bounded VHDL `for` statement with two conditions. The first condition skips all array entries before `current + 1` through a `next` statement that is similar to the ANSI-C `continue` statement. The second condition looks only at entries of the array that are within its size. If the computed lowest index is different than `current` then a swap of values is enabled. Notice that `temp` is not needed since VHDL statements are executed concurrently.

Assertions. The assertions in `selsortproperty` should hold after the search is done. SEBAC encloses the assertions with the conditional `current = size - 1` since the assertions are evaluated only when `selsort` completes its computation.

```
  if (current = size - 1) then
    if (iTest < jTest) then
      assert (a(iTest) < a(jTest) or a(iTest) = a(jTest))
             report "order violated" severity error;
    end if;
    for i in 0 to bound - 1 loop
      if (i < size) then
        if (aTest = a(i)) then
          exit;
```

```
        end if;
      end if;
    end loop;
    assert (aTest = a(i))
           report "entry not found" severity error;
end if;
```

The indices `iTest` and `jTest` are left uninitialized to denote they are nondeterministic and thus capture all possible indexes. The first assertion checks whether two array entries that are indexed by ordered indeces are actually ordered. The second checks whether `aTest`, the value of an arbitrary array entry that was stored before the sorting is still found in the array.

SEBAC currently does not handle recursion implied from branching (`goto` statements). It can be easily extended to unwind them with bounds in a fashion similar to CBMC.

We end up with a hierarchical VHDL design with asserted signals designated as the invariants. We pass the VHDL to SixthSense which, in case of satisfiability, provides a trace that satisfies the inverse of the invariant, otherwise it returns with a proof.

4.5 Results

To evaluate TBV, we compared SEBAC and SPARK coupled with SixthSense, (TBV:SEBAC) and (TBV:SPARK) respectively, against CBMC coupled with SAT (SAT:CBMC).

We took 4 challenging examples of C programs from Chapter 3 of [77]. All the three techniques were able to report helpful error traces within comparable and reasonable resources and small bounds (smaller than 8). We fixed the errors in the 4 programs (for `selsort` we changed Line 9 to `if (a[j] < a[lowestindex]) )` and we report on the time that the analysis took to complete the proofs for a number of bounds in Table 4.2. In the size column, $|V|$ and $|C|$ denote the number of CNF variables and clauses respectively,

Table 4.2: Comparison of SAT:CBMC, TBV:SPARK and TBV:SEBAC.

Selection sort					
Technique	Bound	Size			Time (mins)
SAT:CBMC	16	$\|V\|$=4,753,354	$\|C\|$=18,971,756		114
	32	timed out before generating CNF			∞
TBV:SPARK	64	$\|\mathcal{I}\|$=392	$\|\mathcal{R}\|$=552	|ANDs|=30,334	132
TBV:SEBAC	64	$\|\mathcal{I}\|$=12	$\|\mathcal{R}\|$=397	|ANDs|=92,320	35
Linked list insertion					
Technique	Bound	Size			Time (mins)
SAT:CBMC	16	$\|V\|$=159,370	$\|C\|$=781,39		64
	32	$\|V\|$=1,113,832	$\|C\|$=6,402,757		∞
TBV:SPARK	64	$\|\mathcal{I}\|$=611	$\|\mathcal{R}\|$=984	|ANDs|=22,711	196
TBV:SEBAC	64	$\|\mathcal{I}\|$=18	$\|\mathcal{R}\|$=622	|ANDs|=53,636	54
Linked list removal					
Technique	Bound	Size			Time (mins)
SAT:CBMC	8	$\|V\|$=18,322	$\|C\|$=107,422		26
	16	timed out before generating CNF			∞
TBV:SPARK	64	$\|\mathcal{I}\|$=789	$\|\mathcal{R}\|$=1,240	|ANDs|=27,505	143
TBV:SEBAC	64	$\|\mathcal{I}\|$=24	$\|\mathcal{R}\|$=792	|ANDs|=71,893	74
Memory allocator and deallocator					
Technique	Bound	Size			Time (mins)
SAT:CBMC	16	$\|V\|$=1,411,745	$\|C\|$=4,958,517		93
	32	$\|V\|$=2,818,813	$\|C\|$=9,912,211		∞
TBV:SPARK	32	$\|\mathcal{I}\|$=1,929	$\|\mathcal{R}\|$=1,084	|ANDs|=44,802	156
TBV:SEBAC	32	$\|\mathcal{I}\|$=59	$\|\mathcal{R}\|$=792	|ANDs|=103,433	122

and $|\mathcal{I}|$, $|\mathcal{R}|$, and |ANDs| denote the number of sequential circuit primary inputs, registers, and AND gates respectively as a measure of their complexity [73]. In the time column ∞ denotes a time-out of 360 minutes.

We report on bounds in powers of two since the analysis depends on bit vector encodings of variables and the bounds are on the ranges of these variables. We ran all experiments on a 1.7 GHz Pentium 4 machine with 1 GB memory and used CBMC version 2.5 and SPARK version 1.3.

By default, CBMC iteratively invokes the SAT solver with a larger formula that uses

a bigger loop and recursion unwinding bound until no unwinding assertion is violated. The times reported for CBMC do not include the time needed for CBMC and SAT to perform these iterative checks. We provide CBMC through its command line interface with a tight bound on the loop unwinding to ensure a fair comparison.

4.5.1 Selection sort

The first example is the selection sort algorithm described in Section 4.4 with the bug on Line 9 fixed. We checked both the order of array entries and the data consistency assertions at the same time. CBMC was able to complete the check on the selection sort routine for a bound of 16 on the size of the array in 114 minutes. For bounds bigger than 16, CBMC could not complete before 6 hours. SixthSense was able to complete the check on the sequential circuit generated with SPARK for a bound of 64 in 132 minutes. For the same bound of 64, SixthSense took only 35 minutes to complete the proof on the sequential circuit generated with SEBAC.

4.5.2 Linked list insertion

The second example is a routine that inserts a node into an ordered linked list. List nodes are stored in an array and they are pointed to by indices. Each node has a key value and a next node index. The insert routine takes the index that points to the head of the list and an index that points to the new node to be inserted in the list and returns an index to the new head of the list. We check two properties of the list upon completion of the insertion assuming the properties held before the insertion.

Properties. The first property checks that the value of any node in the list is smaller than or equal to the value of its successor. The second property checks whether the size of the list is consistent (incremented by one after the insertion). CBMC was able to complete the analysis of the list insertion example for a bound of 16 within 164 minutes. However, it took more than 6 hours on larger bounds without completing the analysis. The sequential

analysis completed on the sequential circuit generated with SPARK in 196 minutes. It took only 54 minutes to complete the proof on the circuit generated with SEBAC for the same bound of 64.

4.5.3 Linked list removal

The third example is a routine that removes a node from an ordered list. It takes as input the index to the head of the list, the key value of the node to be deleted, and a writable pointer that should be filled with the index of the deleted node. It returns the index of the new head of the list. We modeled the writable pointer with a static variable since SPARK does not handle pointers. We also relaxed the assertions CBMC generates for pointers to obtain a fair comparison.

Properties. The properties we checked for the removal routine are similar to those we checked for the insertion. We checked whether the order is preserved and whether the size of the list is consistent. It took CBMC 64 minutes to complete the proof with a bound of 16. CBMC generated a CNF formula for a bound of 32 but the analysis did not complete before 6 hours. We completed the proof on the sequential circuit generated with SPARK in 143 minutes for a bound of 64. Sequential analysis performed better with the circuit generated with SEBAC and completed in 74 minutes for the same bound.

4.5.4 Memory allocator and deallocator

Our last example is a memory allocator routine which takes as input a size of desired memory to allocate, returns an index into an array of bytes that represents memory, and signals failure by returning an invalid index (NULL). Internally memory is allocated in blocks of a constant size. The memory allocator considers consecutive groups of memory in the same allocation mode, either allocated or free, as *spans*. It tracks the usage of these blocks in another “in use” array. If a span is free, all its entries in the “in use” array contain a positive value which is the number of blocks in the span. If it is allocated, all entries in the “in use”

array contain a negative number which is the negative of the number of blocks in the span. We also consider a deallocation routine that takes an index to a memory location and frees it.

Properties. We assume that the memory and the "in use" array were initialized to be all free and that the indices passed to the deallocation routine are all aligned correctly with the block sizes. We run an arbitrary but bounded number of allocations and deallocations and then check the "in use" array for consistency with the sizes of allocation and deallocation operations we performed. CBMC completed the proof in 93 minutes for a bound of 16. It could not complete the proof for a bound of 32 in less than 6 hours. TBV:SPARK took 156 minutes to complete the proof for a bound of 32. TBV:SEBAC took 122 minutes to complete the proof for the same bound.

4.5.5 Discussion

The results in Table 4.2 show that by keeping the structure of the program and performing the analysis at the imperative sequential level, we were able to scale the analysis of ANSI-C programs to bounds that are much higher than those achievable by the stateless and flat hierarchy of SAT analysis.

TBV techniques were able to scale to the same bounds and found difficulty going beyond a bound of 64 for both SPARK and SEBAC. However, sequential circuits generated with SEBAC had a clear performance advantage against those generated by SPARK. We attribute the difference to the effort SixthSense needed to make in order to undo the scheduling logic and the additional state elements SPARK introduced to the program for synthesis optimizations and for ease of transformation purposes. Note that the sequential circuits generated by SPARK had more registers and inputs and less logic (AND gates) since SPARK aims at reducing the combinational depth of the circuit.

4.6 Summary

In this chapter we presented SEBAC, a novel static analysis technique for verifying imperative programs. We introduced the use of sequential circuits instead of pure combinational Boolean formulas to encode bounded ANSI-C programs and thus enabled the use of sequential solvers with reduction potentials that have no counterparts in combinational solvers. We were able to show that a bound of 64 is feasible with reasonable resources.

Chapter 5

Discussion

The effort spent on verifying computing systems constitutes half the resources spent in the design cycle of such systems [2]. This created interest in automating software and hardware verification, which grew tremendously in the last few decades. Dynamic analysis is the prevailing approach and it involves testing the system with a large and representative set of concrete inputs. While dynamic techniques are scalable and can be applied to systems of any size, they are incomplete and can not guarantee the absence of bugs and design flaws. On the other hand static analysis techniques study the description of the system and uses rigorous mathematical reasoning to prove claimed properties of the system. Once they complete the analysis, they guarantee that the property in question holds for the system.

Static tools can be classified into two categories. (1.) Theorem proving techniques reason with axioms and rewrite rules to prove a valid logical path from the source description of the system to its claimed properties. These techniques are powerful but often need manual and human interaction to complete proofs on realistic systems. (2.) Automatic model checking techniques work on the semantics of the description of the system and its specifications.

This dissertation presented novel algorithms and techniques that improve the automatic static verification of computing systems.

Designers have to formally describe designs and properties before presenting them to static verification techniques. They use high-level description languages or logics that typically get transformed by analysis techniques into propositional Boolean representations. The latter representations are addressed by model checkers empowered by BDD or SAT based solvers as well as other techniques such as partial reduction and predicate abstraction.

The high-level description languages can be classified as either imperative or declarative. A declarative language, such as the first order logic with transitive closure Alloy, declares the system objects and describes the relations between these objects through a set of constraints and then claims a property over the constrained system. They model the conditions that are true when a computation occurs rather than describing the steps in the operation. An Imperative language, such as the C programming language, describes algorithmically the steps of a computation and claims a property that describes the correctness of the computation.

These languages are often undecidable and thus model checkers, such as Alloy Analyzer and CBMC, prove the correctness of the claimed properties for a certain bound on the ranges of the variables or iterations of the loops in a program. The state of the art tools find a problem handling realistic systems with realistic enough bounds. For example, presented with a check of a tree integrity property of all trees up to 8 nodes, the Alloy Analyzer failed to generate a Boolean CNF formula that can be passed to the back end SAT solver.

5.1 Summary of contributions

To extend the applicability of static analysis to a wider class of designs as well as to check more sophisticated properties and gain more confidence in the results, we need to scale the analysis to significantly larger bounds.

This dissertation bases its contributions on the following observations.

1. High-level insight is very instrumental in reducing designs to a size that is amenable

to static verification techniques. This is often lost in the transformation to low-level propositional Boolean formulas. This dissertation exploits the structure of the system under test when it formulates the Boolean problem passed to the back-end solver.

2. Sequential circuits are more succinct than propositional Boolean logic in representing designs of iterative or computational nature. This dissertation reduces the size and complexity of the problem by encoding designs into sequential circuits where appropriate.

3. Sequential solvers exploit numerous powerful reduction and decision techniques that have no counterparts in propositional logic. The work in this dissertation enabled and leveraged the power and synergy of different sequential reduction and decision techniques to solve the problem.

4. Declarative and imperative programs can both benefit from static sequential analysis since they have constructs that are concisely encoded with stateful structures and hierarchies that can be modeled in sequential circuits. This dissertation evaluated the static sequential analysis approach and showed that it can check programs for bounds order of magnitudes higher than those feasible with current techniques such as Alloy Analyzer and CBMC.

5.2 Future work

In the future we plan to explore how our approach of sequential encoding may be extended to other logic specifications such as S1S [59], and Presburger Arithmetic (PA) [86]. We are also interested in investigating security applications of PA program equivalence detection.

We also plan to study the use of Alloy as a property specification language for C programs combining the benefits of declarative and imperative languages.

It has long been known that unquantified equality logic has a finite model property, specifically, a formula $\Phi(x_0, ..., x_n)$ in this logic is satisfiable iff it is satisfiable in a

universe of cardinality n. We plan to express equality theory formulas as netlists and use communication complexity arguments to develop tighter bounds for equality logic in many cases of interest.

Bibliography

[1] "The economic impacts of inadequate infrastructure for software testing," in *National Institute of Standards and Technology, Planning report 02-3*, May 2002.

[2] E. Clarke, J. O. Brumberg, and D. A. Peled, *Model Checking*. MIT Press, 1999.

[3] L. Copeland, *A Practitioner's Guide to Software Test Design*. Artech House, 2003.

[4] A. Aziz, *Formal Methods in VLSI System Design*. PhD thesis, University of California, Berkley, 1996.

[5] H. B. Enderton, *A Mathematical Introduction to Logic*. Academic Press, December 2000.

[6] M. Kaufmann and J. S. Moore, "An industrial strength theorem prover for a logic based on common lisp," *Software Engineering*, vol. 23, no. 4, pp. 203–213, 1997.

[7] S. Fine and A. Ziv, "Coverage directed test generation for functional verification using bayesian networks," in *Design Automation Conference*, 2003.

[8] R. Majumdar and K. Sen, "Hybrid concolic testing," in *International Conference on Software Engineering*, 2007.

[9] K. L. McMillan, *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[10] M. Sheeran, S. Singh, and G. Stalmarck, "Checking safety properties using induction and a SAT-solver," in *Formal Methods in Computer-Aided Design*, pp. 108–125, Nov. 2000.

[11] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Tools and Algorithms for Construction and Analysis of Systems*, pp. 193–207, March 1999.

[12] L. Momtahan, "Towards a small model theorem for data independent systems in alloy.," *Electronic Notes in Theoretical Computer Science*, 2005.

[13] D. E. Long, *Model Checking, Abstraction and Compositional Verification*. PhD thesis, Carnegie Mellon University, July 1993.

[14] J. Baumgartner, *Automatic Structural Abstraction Techniques for Enhanced Verification*. PhD thesis, University of Texas, Dec. 2002.

[15] K. S. Namjoshi and R. P. Kurshan, "Syntactic program transformations for automatic abstraction," in *Computer-Aided Verification*, pp. 435–449, July 2000.

[16] H. Jain, D. Kroening, N. Sharygina, and E. Clarke, "Word level predicate abstraction and refinement for verifying rtl verilog," in *Design Automation Conference*, June 2005.

[17] Z. Andraus and K. Sakallah, "Automatic abstraction and verification of verilog models," in *Design Automation Conference*, June 2004.

[18] D. Jackson, *Alloy 3.0 Reference Manual*, May 2004. http://alloy.mit.edu/reference-manual.pdf.

[19] I. A. Shlyakhter, *Declarative Symbolic Pure-Logic Model Checking*. PhD thesis, MIT, February 2005.

[20] S. Lahiri and S. Seshia, "The uclid decision procedure," in *Computer Aided Verification*, 2004.

[21] J. Elgaard, N. Klarlund, and A. Mller, "Mona 1.x: new techniques for ws1s and ws2s," in *Computer Aided Verification*, Springer Verlag, 1998.

[22] R. K. Brayton et al., "VIS: A system for verification and synthesis," in *Computer-Aided Verification*, July 1996.

[23] M. Chiodo, P. Guisto, A. Jurecska, L. Lavagno, H. Hsieh, K. Suzuki, A. L. Sangiovanni-Vincentelli, and E. Sentovich, "Synthesis of software programs for embedded control applications," in *Design Automation Conference*, 1995.

[24] H. Mony et al., "Scalable automated verification via expert-system guided transformations," in *Formal Methods in Computer-Aided Design*, Nov. 04.

[25] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, March 2004.

[26] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *ACM Design Automation Conference*, June 2001.

[27] E. Goldberg and Y. Novikov, "Berkmin: A fast and robust sat solver," in *Design Automation and Test in Europe*, 2002.

[28] J. P. Marques-Silva and K. A. Sakallah, "GRASP: a search algorithm for propositional satisfiability," *IEEE Transactions on Computers*, vol. 48, no. 5, pp. 506–521, 1999.

[29] R. Bryant, "Graph-based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, vol. C-35, pp. 677–691, Aug. 1986.

[30] S. K. Lahiri, E. Randal, E. Bryant, and B. Cook, "A symbolic approach to predicate abstraction," in *Computer-Aided Verification*, pp. 141–153, July 2003.

[31] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Design Automation Conference*, June 2001.

[32] R. J. Bayardo Jr. and R. C. Schrag, "Using CSP look-back techniques to solve real world SAT instances," in *Proceedings of the National Conference on Artificial Intelligence*, 1997.

[33] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah, "Solving difficult sat instances in the presence of symmetry," in *Design automation conference*, ACM Press, 2002.

[34] Z. Fu, Y. Yu, and S. Malik, "Considering circuit observability don't cares in cnf satisfiability," in *Proceedings of Design, Automation and Test in Europe*, pp. 1108–1113, 2005.

[35] Q. Zhu, N. Kitchen, A. Kuelhman, and A. Sangiovanni-Vincentelli, "Sat sweeping with local observability dontcares," in *Design Automation Conference*, July 2006.

[36] S. A. Edwards, "The challenges of hardware synthesis from C-like languages," in *Design Automation and Test in Europe*, 2005.

[37] P. Ashenden, *The Designers Guide to VHDL*. Morgan Kaufmann, 2002.

[38] G. De Micheli, "Hardware synthesis from C/C++ models," in *Design Automation and Test in Europe*, Mar. 1999.

[39] A. Kuehlmann and J. Baumgartner, "Transformation-based verification using generalized retiming," in *Computer-Aided Verification*, July 2001.

[40] H. Mony, J. Baumgartner, V. Paruthi, and R. Kanzelman, "Exploiting suspected redundancy without proving it," in *Design Automation Conference*, ACM Press, 2005.

[41] A. Kuehlmann, M. Ganai, and V. Paruthi, "Circuit-based Boolean reasoning," in *Design Automation Conference*, pp. 232–237, June 2001.

[42] P. Bjesse and K. Claessen, "SAT-based verification without state space traversal," in *Formal Methods in Computer-Aided Design*, November 2000.

[43] A. Aziz, T. Shiple, V. Singhal, R. Brayton, and A. Sangiovanni-Vincentelli, "Formula Dependent Equivalence for Compositional CTL Model Checking," *Journal of Formal Methods in System Design*, vol. 21, no. 2, pp. 193–224, 2002.

[44] P. Bjesse and A. Boralv, "DAG-aware circuit compression for formal verification," in *Int'l Conference on Computer-Aided Design*, Nov. 2004.

[45] K. L. McMillan, *Interpolation and SAT-Based Model Checking*. Springer Berlin / Heidelberg, 2003.

[46] D. Wang, *SAT based Abstraction Refinement for Hardware Verification*. PhD thesis, Carnegie Mellon University, May 2003.

[47] I.-H. Moon, G. D. Hachtel, and F. Somenzi, "Border-block triangular form and conjunction schedule in image computation," in *Formal Methods in Computer-Aided Design*, Nov. 2000.

[48] P.-H. Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor, and J. Long, "Smart simulation using collaborative formal and simulation engines," in *Int'l Conference on Computer-Aided Design*, Nov. 2000.

[49] J. Baumgartner, A. Kuehlmann, and J. Abraham, "Property checking via structural analysis," in *Computer-Aided Verification*, July 2002.

[50] F. Zaraket, A. Aziz, and S. Khurshid, "Sequential circuits for relational analysis," in *International Conference on Software Engineering*, May 2007.

[51] S. Gupta, R. K. Gupta, N. D. Dutt, and A. Nicolau, "Coordinated parallelizing compiler optimizations and high-level synthesis," *ACM Transactions on Design Automation of Electronic Systems*, October 2004.

[52] T. Kropf, *Introduction to Formal Hardware Verification*. Springer-Verlag, 1998.

[53] S. H. abd Tsutomu Sasao, ed., *Logic Synthesis and Verification*. Kluwer Academic Publishers, 2003.

[54] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[55] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic model checking: 10^{20} states and beyond," *Information and Computation*, vol. 98, no. 2, 1992.

[56] A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification," *IEEE Transactions on Computer-Aided Design*, vol. 21, no. 12, 2002.

[57] F. Zaraket, J. Baumgartner, and A. Aziz, "Scalable compositional minimization via static analysis," in *International Conference on Computer Aided Design*, Nov. 2005.

[58] R. K. Brayton and C. McMullen, "The Decomposition and Factorization of Boolean Expressions," in *International Symposium on Circuits and Systems*, May 1982.

[59] A. Aziz, F. Balarin, R. Brayton, and A. Sangiovanni-Vincentelli, "Sequential Synthesis Using `S1S`," *IEEE Trans. Computer-Aided Design*, Oct. 2000.

[60] A. Saldanha, A. R. Wang, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Multi-Level Logic Simplification using Don't Cares and Filters," in *Design Automation Conference*, pp. 277–282, 1989.

[61] R. P. Kurshan, *Automata-Theoretic Verification of Coordinating Processes*. Princeton University Press, 1993.

[62] Z. Fu and S. Malik, "Extracting logic circuit structure from conjunctive normal form descriptions," in *Proceedings of International Conference on VLSI Design*, January 2007.

[63] J. Hopcroft and J. Ullman, *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[64] D. Box, *Essential COM*. Addison Wesley, 1998.

[65] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley, "The design and implementation of an intentional naming system," in *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, (Kiawah Island), Dec. 1999.

[66] G. Dennis, "TSAFE: Building a trusted computing base for air traffic control software," Master's thesis, Massachusetts Institute of Technology, 2003.

[67] D. Jackson and M. Jackson, *Software Abstractions: Logic, Language, and Analysis*.

[68] M. O. Rabin, "Decidability of second-order theories and automata on infinite trees," *Transactions of the American Mathematical Society*, 1969.

[69] N. Immerman, A. Rabinovich, T. Reps, S. Sagiv, and G. Yorsh, "The boundary between decidability and undecidability for transitive-closure logics," in *Computer Science Logic*, 2004.

[70] D. Jackson, "Automating first-order relational logic," in *ACM-SIGSOFT Symposium on Foundations of Software Engineering*, 2000.

[71] A. Seawright and F. Brewer, "High-level symbolic construction technique for high performance sequential synthesis," in *Design Automation Conference*, 1993.

[72] M. T. Oliveira and A. J. Hu, "High-level specification and automatic generation of IP interface monitors," in *Design Automation Conference*, 2002.

[73] V. Paruthi and A. Kuehlmann, "Equivalence checking combining a structural SAT-solver, BDDs, and simulation," in *Proceedings of the Int'l Conference on Computer Design*, pp. 459–464, Sept. 2000.

[74] G. Holzmann, "The model checker SPIN," in *IEEE Transactions on Software Engineering*, May 1997.

[75] W. Visser, K. Havelund, G. Brat, and S.-J. Park, "Model checking programs," in *Automated Software Engineering Journal*, April 2003.

[76] E. Torlak and D. Jackson, "Kodkod: A relational model finder," in *Tools and Algorithms for Construction and Analysis of Systems*, March 2007.

[77] A. Barr, *Find The Bug: A Book of Incorrect Programs*. Addison-Wesley, 2002.

[78] S. Haynal, *Automata-Based Symbolic Scheduling*. PhD thesis, University of California, Santa Barbara, 2000.

[79] N. Weste and D. Harris, *CMOS VLSI Design - A Circuits and Systems Perspective.* Addison-Wesley Publishing Company, 2005.

[80] K. K. Parhi, *VLSI Digital Signal Processing Systems: Design and Implementation*. Wiley-Interscience, January 1999.

[81] E. Clarke, D. Kroening, and K. Yorav, "Behavioral consistency of c and verilog programs using bounded model checking," in *Design Automation Conference*, 2003.

[82] International Organization for Standardization, *ISO/IEC 9899:1999: Programming languages—C*. 1999.

[83] M. J. C. Gordon and T. F. Melham, eds., *Introduction to HOL: a theorem proving environment for higher order logic*. New York, NY, USA: Cambridge University Press, 1993.

[84] M. Norrish, "C formalized in HOL," tech. rep., University of Cambridge Computer Laboratory, December 1998.

[85] X. Wang and E. P. Stabler, "Formalization of vhdl synthesis procedure in higher-order logic," in *Proceedings of International Tutorial and Workshop on the HOL Theorem Proving System and its Applications*, 1991.

[86] T. Shiple, J. Kukula, and R. Ranjan, "A comparison of presburg engines for efsm reachability," in *Computer-Aided Verification*, June 1998.

Vita

Fadi Zaraket was born to Abdul-Majid Zaraket and Isaaf Sabbagh, in Markaba, Lebanon in 1974. He enjoys playing chess and soccer and writes Arabic poetry and short stories. Fadi joined the Faculty of Engineering and Architecture in the American University of Beirut in 1992 and received his Bachelor of Engineering degree in Computer and Communication Engineering in July 1996 and his Masters of Engineering degree in February 2001. Fadi works in the industry on building tools for functional and static verification and visualization of hardware logic designs. His research interests are in automating logic analysis and applying it to large and complex designs.

Permanent Address: 15027 Jacks Pond Rd
Austin TX, 78728

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.