

AMERICAN UNIVERSITY OF BEIRUT

Verification of Software and Embedded Systems
using AIG Solvers

by

MOHAMAD ALI NOUREDDINE

A thesis

submitted in partial fulfillment of the requirements
for the degree of Masters in Engineering
to the Department of Electrical and Computer Engineering
of the Faculty of Engineering and Architecture
at the American University of Beirut

Beirut, Lebanon
May 2014

AMERICAN UNIVERSITY OF BEIRUT

Verification of Software and Embedded Systems using AIG Solvers

by

MOHAMAD ALI NOUREDDINE

Approved by:

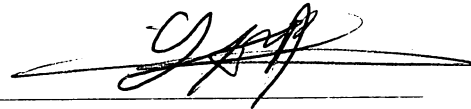
Dr. Fadi Zaraket, Assistant Professor
Electrical and Computer Engineering

Advisor



Dr. Louay Bazzi, Associate Professor
Electrical and Computer Engineering

Member of Committee



Dr. Wassim Masri, Associate Professor
Electrical and Computer Engineering

Member of Committee



Date of thesis defense: May 8th, 2014

Acknowledgments

Foremost, I would like to thank God for giving me the chance to continue my graduate studies at the American University of Beirut, and for allowing me to enjoy the support of all the wonderful people I have met during this journey. It is also through the divine guidance of the Quoran and the Holy prophet Muhammad and his family that I was able to keep faith throughout the hardships and obstacles that I have faced.

Imam Zainul Abideen (a.s.) says in his Treatise of Rights:

The right of the one who trains you through knowledge is magnifying him, respecting his sessions, listening well to him, and attending to him with devotion. You should not raise your voice toward him. You should never answer anyone who asks him about something, in order that he may be the one who answers. You should not speak to anyone in his session nor speak ill of anyone with him. If anyone ever speaks ill of him in your presence, you should defend him.

I am forever indebted to my thesis adviser, Fadi Zaraket, for the continuous guidance and help he provided me with during my period of graduate studies. His approach to logic and formal methods is what attracted me towards this field and kept me interested in it. It is with his encouragement and accurate advice that I was able to tackle the problems I faced in my research. I would also like to extend my gratitude to Fadi for the confidence he put in me and for allowing me to extend the work he started in his PhD.

I am also full of gratitude to my thesis committee members, Louay Bazzi and Wassim Masri, for their constructive comments and advice. I would also like to thank them for making my thesis an enjoyable and instructive experience. I would also like to thank Mohamad Jaber for the help that he provided me with in my work, especially for investing his time in a common research project, and for assisting me in finding an internship opportunity.

Furthermore, I could not have completed this stage of my academic life without the unconditional love, constant care and support, and prayer of my beloved fiancé Samah Karim. Thank you for being by my side and for listening to my constant nagging in the times where I faced obstacles in my research. I couldn't have imagined a better companion for me in my journey and my life.

Finally, key to my academic success is the support of my father, Ali, my mother, Sonia, and my brother, Houssein. I owe all of the goals I was able to reach in my life to their constant care and encouragement. I would also like to thank them for the huge faith that they had in me all along the way, which has pushed me to reach my production limits and beyond.

AN ABSTRACT OF THE THESIS OF

MOHAMAD ALI NOUREDDINE

for

Master of Engineering

Major: Electrical and Computer Engineering

Title: Verification of Software and Embedded Systems using AIG Solvers

It is critical for software and hardware developers to design correct and reliable systems. In particular, safety critical systems such as medical equipment, navigation control and targeting devices do not tolerate defects in their logical components. Static analysis techniques are used to check and prove correctness of logic components with respect to formal specifications. In particular, ABC is a model checker that takes an And-Inverter-Graph (AIG) circuit, a directed acyclic graph with two input AND gates, inverters and memory elements, reduces it using synthesis algorithms, and checks it for correctness using proof algorithms. Existing techniques transform software programs and embedded system design components into Conjunctive Normal Form (CNF) formulae and Symbolic Model Verifier (SMV) code, and use satisfiability (SAT) solvers and symbolic model checkers, respectively, to check their validity within a user specified finite domain. These techniques often fail to scale well with the increasing size of systems and with larger finite domains.

In this work, we explore the use of AIG solvers to address the verification of software and embedded systems subject to bounds on the data width of their variables. $\{P\}S\{Q\}$ translates imperative logic systems, written in a C-like language, into AIG. $BIP\{I\}$ translates an embedded system, written within the Behaviour-Interaction-Priority (BIP) framework, into AIG. Both methods use the ABC AIG solver to reduce the generated AIG circuits using sequential synthesis algorithms, and then check them for validity. The solver either (1) proves the specifications valid within the finite domain, (2) generates a counter example and reports it to the developer for debugging, or (3) reaches its computational bounds before making a decision. We evaluated $\{P\}S\{Q\}$ against a set of array and list manipulation algorithms, and various benchmarks obtained from the second competition on software verification (SV-Comp'13). Results show that $\{P\}S\{Q\}$ reaches bounds higher than those possible with the CBMC bounded model checker. It was also able to rank amongst the top three tools in the software verification competition. We also evaluated $BIP\{I\}$ against two benchmarks, an Automatic Teller Machine (ATM) system and the Quorum consensus protocol. $BIP\{I\}$ surpasses the NuSMV model checker on both designs.

Contents

Contents	vii
List of Figures	ix
List of Tables	x
1 Introduction	1
1.1 Advantages of sequential circuits	3
2 Background	5
2.1 Sequential circuits	5
2.2 The <i>CAIG</i> component language	6
2.3 ABC sequential solver	7
2.3.1 Structural Register Sweep (SRS)	7
2.3.2 Signal Correspondence (Scorr)	8
2.3.3 Rewriting	8
2.3.4 Retiming	8
2.3.5 Property Directed Reachability (Pdr)	8
2.3.6 Temporal Induction	9
2.3.7 Interpolation	9
3 $\{P\}S\{Q\}$: Imperative Programs to AIG	11
3.1 Limitations of translation to CNF	13
3.2 The <i>Tiny</i> imperative language	14
3.3 Translation to AIG	15
3.3.1 Preprocessing	15
3.4 Transformation to <i>CAIG</i>	20
3.5 <i>CAIG</i> to AIG	22
4 <i>BIP</i>$\{I\}$: BIP to AIG	26
4.1 BIP - Behavior Interaction Priority	26
4.1.1 Component-based Construction	27
4.2 BIP to <i>CAIG</i>	29
4.3 Illustrative Example	31

5	Implementation	36
5.1	$\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$	36
5.2	$BIP\{\mathcal{I}\}$	39
6	$\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ Results	42
6.1	Searching: $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ and CBMC	42
6.2	Standard benchmarks	44
6.3	SV-COMP 2013 benchmarks	46
7	$BIP\{\mathcal{I}\}$ Results	49
7.1	The ATM benchmark	49
7.2	The Quorum protocol	50
8	Related work	53
8.1	Verification of software programs	53
8.2	Verification of embedded systems	54
9	Conclusion	56
	Bibliography	57

List of Figures

2.1	<i>CAIG</i> component language grammar	7
3.1	(a) Array search program, and (b) equivalent array search program with program counter	12
3.2	The <i>Tiny</i> imperative language	16
3.3	Transformation function ρ on function calls	17
3.4	Transformation function ρ on universal quantifier	19
3.5	Post processing of a <i>CAIG</i> \mathcal{C}	23
4.1	Traffic light controller BIP system	26
5.1	Architecture of $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$	36
5.2	Example source code	37
5.3	<i>BIP</i> $\{\mathcal{L}\}$'s command line options	41
6.1	The linear search algorithm with pre and post conditions	43
6.2	The binary search algorithm with pre and post conditions	44
7.1	Modeling of ATM system in BIP	49
7.2	Visualization of a counter example using Gtkwave	52

List of Tables

2.1	ABC algorithms	10
4.1	Sample of $\mathcal{BIP}\{\mathcal{I}\}$ execution	32
5.1	Summary of $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ commands	40
6.1	$\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ and CBMC comparison	45
6.2	Results of standard benchmarks	46
6.3	The properties checked for the standard benchmarks	47
6.4	SVCOMP'13 results	48
7.1	ATM results	50
7.2	Quorum results	52

Chapter 1

Introduction

It is critical for software and hardware developers to design correct and reliable systems. In particular, safety critical systems such as medical equipment, navigation control and targeting devices do not tolerate defects in their logical components. A logical defect might lead to severe consequences including loss of human life. A loss of precision due to a conversion from 64-bits into 16-bits of an integer value, caused the Ariane-5 missile to crash only 16 seconds after it has been launched. Additionally, a software bug in the control of a radiation therapy machine, “Therac-25”, has led to the death of six cancer patients between 1985 and 1987 [1].

In addition, information technology (IT) systems have made their way into several aspects of modern life. Smart phones, TVs, personal computers, laptops, banking, money transfer and e-shopping rely on IT systems. Logic defects in such systems can lead to loss of service, as well as other damages. Intel had to recall its faulty Pentium processors, that had a bug in the floating point unit, for a cost of 475 million US dollars [2].

Researchers introduced techniques to (1) accurately specify the requirements of logic components, (2) validate that the implementation of a logic component respects the specifications for a finite bounded domain, and (3) prove that a logic component respects its specifications for unbounded domains. *Dynamic analysis* techniques exercise the logical component for a given set of inputs, e.g. test cases, and check the results against a correctness predicate or an oracle [1]. Exhaustive testing is impossible in practical scenarios, because it involves exercising a practically infinite number of test cases [3].

Static analysis techniques are used to check and prove correctness of logic components with respect to formal specifications. Tools exist that synthesize, optimize, and check sequential circuits. ABC [4] is an industrial strength academic sequential synthesis and verification tool. It takes as input an And-Inverter-Graph (AIG) representation of a sequential circuit, reduces it using synthesis and reduction algorithms, and checks it for correctness using proof algorithms. An AIG is a directed acyclic graph with two input AND gates, inverters and memory elements. Since AND gates and inverters (i.e. NAND gates) are functionally complete, the restriction of logic gates to ANDs and inverters does not affect expressiveness.

Software systems are usually designed in high level programming languages

such as C, C++ and JAVA. Software verification tools such as CBMC [5] take as input software programs and check them for memory safety, array access safety, and user defined assertions (invariants). Given bounds on the unwinding depth of the loops, and on range of variables in the program, CBMC unfolds the program into a Boolean *Conjunctive Normal Form* (CNF) formula that asserts the specified properties. CBMC then uses Boolean *satisfiability* (SAT) solvers such as MiniSat [6] to check for correctness. CBMC verification tasks often fail to scale well with the increase in the size of the problem; specifically, with the loop unwinding bound and the complexity of the program.

Furthermore, in recent years, the application area of embedded systems has witnessed a large expansion, especially with the emergence of automotive electronics and mobile and control devices. Computations in embedded systems are subject to several physical and architectural constraints that render the separation between software and hardware design impractical [7]. The Behavior-Interaction-Priority (BIP) framework is a *Component-Based System* (CBS) design framework that uses a dedicated language and tool-set in order to support a rigorous and layered design flow for embedded systems. It allows to build complex systems by coordinating the behavior of a set of atomic components [8]. In order to check CBS, the BIP framework uses: (1) DFinder [9], a compositional and incremental verification tool-set, and (2) the NuSMV [10] model checker. However, DFinder does not handle data transfer between individual BIP components, and does not support checking for properties other than deadlock freedom. Additionally, for complex systems, NuSMV often suffers from the state space explosion problem [11], and fails to perform its verification tasks.

In this thesis, we present two techniques with supporting tools to address the verification of both software systems and CBSs: (1) $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ and (2) $\mathcal{BIP}\{\mathcal{I}\}$, respectively. $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ is a tool that takes an imperative program \mathcal{S} with a specification, a precondition and postcondition pair $(\mathcal{P}, \mathcal{Q})$, and checks whether \mathcal{S} satisfies the specification. This check is performed within a bound b on the domain of the program and specification variables ($\mathcal{S} \models (\mathcal{P}, \mathcal{Q})|_b$); i.e. when the bounded inputs of \mathcal{S} satisfy \mathcal{P} , the outputs of \mathcal{S} satisfy \mathcal{Q} . The program is written in a subset of C++ that includes integers, arrays, loops, and recursion, extended with a `do_together` construct. The specification consists of a precondition \mathcal{P} and a postcondition \mathcal{Q} , both written in *first order logic* (FOL). Our method translates the problem $\mathcal{S} \models (\mathcal{P}, \mathcal{Q})|_b$ into an equisatisfiable AIG, and passes the AIG to ABC for sequential synthesis and model checking. If successful, the check implies that the \mathcal{S} satisfies its specification pair $(\mathcal{P}, \mathcal{Q})$. Otherwise, it returns a counterexample and enables the developer to visualize it using the GtkWave [12] waveform viewer for debugging.

$\mathcal{BIP}\{\mathcal{I}\}$ is a tool that targets two design goals of the BIP framework: (1) verification and (2) code generation. Similarly to $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$, it takes an input BIP system with a set of optional specifications, and translates it in an equisatisfiable AIG. It then uses the ABC model checker to perform synthesis and verification. Additionally, unlike the BIP code generator used in [8] that uses a generic implementation of the BIP engine, $\mathcal{BIP}\{\mathcal{I}\}$ generates an AIG system with a customized engine that is able to execute multiple atomic component transitions simultaneously.

1.1 Advantages of sequential circuits

We formally define sequential circuits in Chapter 2; 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 [13].

There are two key advantages to compiling systems into sequential circuits:

Advantage 1 Sequential encodings are much more succinct than pure combinational SAT formulae. They are imperative and state-holding while CNF formulas, for example, are declarative and state-free. For example, they can naturally represent the execution of quantifiers and loops without the need for unrolling them. Moreover, they can store and reuse intermediate results in local variables. 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 system as an invariant check on a sequential circuit allows us to make use of a number of powerful automated analysis techniques that we discuss in Chapter 2 and that have no counterpart in CNF analysis.

$\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ and $BIP\{\mathcal{I}\}$ use ABC [4] to automatically check invariants in AIGs. ABC is a transformation-based verification (TBV) [14] framework that encompasses reduction and abstraction techniques such as retiming [15], redundancy removal [16, 17, 18, 19], logic rewriting [20], interpolation [21], and localization [22]. It operates on AIGs; 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 symbolic model checking, bounded model checking, induction, interpolation, circuit SAT solving, and target enlargement [23, 24, 25, 26, 27].

In this thesis we make the following contributions.

- We encode an imperative program \mathcal{S} with a first order logic specification $(\mathcal{P}, \mathcal{Q})$ into an AIG with an invariant that is stuck to *true* iff \mathcal{S} satisfies \mathcal{Q} for all inputs that satisfy \mathcal{P} within a given bound on the range of the program variables.
- We use a program counter semantics to encode the \mathcal{S} and $(\mathcal{P}, \mathcal{Q})$ into an AIG. We use that to encode and compute a termination guarantee check within a bound on the number of iterations or recursive calls a program can make. We use the termination guarantee bound to prove run time efficiency of given algorithms.
- We encode a BIP system into an AIG with an invariant that is stuck to *true* iff the system is deadlock free, or satisfies a set of specifications. The generated AIG system will have its own customized engine that is able to execute multiple transitions simultaneously.
- We allow for simulation of the BIP system by generating a C++ code, while allowing for optional user control over the interactions to execute.

- We use the ABC sequential circuit verification framework to check the generated AIGs, and we check systems and programs that are orders of magnitude higher than those possible with the other techniques.
- We implement our methods in two tools: $\{P\}S\{Q\}$ and $BIP\{I\}$.
- We provide our tools and benchmarks online ¹.

The rest of this thesis is organized as follows. Chapter 2 gives an overview of the preliminary information needed throughout this thesis. Chapters 3 and 4 present the techniques used in $\{P\}S\{Q\}$ and $BIP\{I\}$, respectively. Chapter 5 highlights the main implementation details and gives a brief user manual for each of the two tools. We present our results in chapters 6 and 7. We discuss relevant related work in chapter 8 to finally conclude in chapter 9.

¹<http://webfea.fea.aub.edu.lb/fadi/dkwb/doku.php?id=sa>

Chapter 2

Background

In this chapter, we present formalisms used throughout this thesis. A reader well-formed in logic verification and sequential circuits may wish to skip this section, using it only as a reference.

Let $V = \{v_1, v_2, \dots, v_m\}$ be a set of scalar variables and $A = \{a_1, a_2, \dots, a_n\}$ be a set of array variables.

Definition 1 (terms). A *term* is either a variable $v \in V$, a constant $c \in \mathbb{Z}$, or an indexed array variable of the form $a[t]$ which denotes the t^{th} element of a where $a \in A$ and t is a term. Arithmetic expressions of the form $-t, t_1 + t_2, t_1 - t_2, t_1 * t_2, t_1/t_2, t_1 \% t_2$ are all terms where t, t_1, t_2 are terms and $-, +, *, /,$ and $\%$ denote the subtraction, addition, multiplication, division and remainder operations, respectively.

Definition 2 (Boolean term). A constant from the set $\mathbb{B} = \{true, false\}$ is a *Boolean term*. The expressions $t_1 < t_2, t_1 \leq t_2, t_1 > t_2, t_1 \geq t_2, t_1 == t_2$ are all Boolean terms where t_1, t_2 are terms and $<, \leq, >, \geq,$ and $==$ denote smaller, less than or equal, bigger than, bigger than or equal, and equal, respectively. The expressions $b_1 \& \& b_2, b_1 ||| b_2, !b_1, - >, ==$ are all Boolean terms where b_1, b_2 are Boolean terms and $\& \&, |||, !, - >, ==$ denote logical conjunction, disjunction, negation, implication, and equivalence, respectively.

Definition 3 (First order logic formula). A Boolean term is a *first order formula*. A quantified formula of the form $Qq.b(q)$, where $Q \in \{\forall, \exists\}$ is either a universal or existential quantifier, q is a quantified variable, and $b(q)$ is a first order formula with q as a free variable.

2.1 Sequential circuits

The ABC solver operates on an sequential circuit representation of a program.

Definition 4 (Sequential circuit). A *sequential circuit* is a tuple $((V, E), G, O)$. 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 5 (Fanins). We define the direct *fanins* 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 .

The ABC solver reasons about *And-Inverter-Graphs (AIG)* which are acyclic sequential circuits with only AND gates and inverters. All AND gates are restricted to have 2 fanins. Since AND gates and inverters are 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) and gates should have two fanins. The initial-value functions of R shall have no registers in their support. All sequential circuits we consider will be well-formed.

Definition 6 (State). A *state* is a Boolean valuation to vertices in R .

Definition 7 (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 $t(v, i)$.

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

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.

2.2 The *CAIG* component language

```

1 component: decl wiredef init 'while(true)' '{' next '}'
2 type: bool | int | bool '['NUM'] | int '['NUM']
3 declaration: wire type ID ';' | type ID ';'
4
5 decl: declaration+
6 wiredef: (target = expr ';' ) *
7
8 init: '@do_together' '{' (target = expr ';' ) * '}'
9 next: '@do_together' '{' (target = expr ';' ) * '}'
10 target: ID | ID '['expr']
11 expr: expr? expr : expr

```

Figure 2.1: *CAIG* component language grammar

The grammar in Figure 2.1 describes *CAIG*, a high level imperative language that describes a sequential circuit. An *CAIG* program starts with a list of declarations of wire, register, and array variables. Wires are defined in a list of assignment statements in the `wiredef` block. Each wire can be the target at most one assignment statement. If a wire is not assigned, then it is left as a free input to the circuit.

The `init` list of statements assigns initial values for the register variables. All assignment statements within the `init` block execute simultaneously as indicated with the `do_together` keyword. Similarly, the `next` list of statements updates the values of the register variables.

Each assignment statement has a left hand side target term which is either a variable or an access operator to an array element. The right hand side of an assignment is a combinational expression that is either a term (from Definition 1) or a ternary choice expression. The ternary choice (`a?b:c`) returns `b` if `a` is `true` and `c` otherwise.

2.3 ABC sequential solver

ABC is an open source sequential circuit solver that operates on a sequential circuit in AIG format and checks the satisfiability of a designated output gate therein. ABC applies several reduction and abstraction techniques to simplify and decompose the problem into smaller problems. It then calls decision techniques to decide the simplified problems. In what follows we discuss some of the techniques that are briefly listed in Table 2.1.

2.3.1 Structural Register Sweep (SRS)

SRS detects registers that are stuck-at-constant and eliminates them from a given sequential AIG circuit. The technique starts by zeroing up all initial values of registers in the circuit. It then uses the ternary simulation algorithm in order to detect stuck-at-constant registers. The algorithm starts from the initial values of the registers and simulates the circuit using `x` values for the circuit's primary inputs. The simulation algorithm stops when a new ternary state is equal to a previously computed ternary state. In this case, any register having the same constant value at each reachable ternary state will be declared to be stuck-at-constant and thus eliminated. The struc-

tural sweeping algorithm stop when no further reduction in the number of registers is possible [28].

2.3.2 Signal Correspondence (Scorr)

Scorr uses k -step induction in order to detect and merge sets of classes of sequentially-equivalent nodes [28]. The base case for this algorithm is that the equivalence between the classes holds for the first k frames, and the inductive case is that given the base case, starting from any state, the equivalence holds in the $(k + 1)^{st}$ state. Key to the signal correspondence algorithm is the way the candidate equivalences are assumed for the base case. Abc implements speculative reduction, originally presented in [35], which merges, but does not remove, any node of an equivalence class onto its representative, in each of the first k time frames. Instead of removing the merged node, a constraint is added to assert that the node and its representative are equal. This technique is claimed to decrease the number of constraints added to the SAT solved for induction.

2.3.3 Rewriting

Rewriting aims at finding nodes in a Directed Acyclic Graph (DAG) where by replacing subgraphs rooted at these nodes by pre-computed subgraphs can introduce important reductions in the DAG size, while keeping the functionality of these nodes intact. The algorithm traverses the DAG in depth-first post-order and gives a score for each root node. The score represents the number of nodes that would result from performing a rewrite at this node. If a rewrite exists such that the size of the DAG is decreased, such a rewrite is performed and scores are recomputed accordingly. Rewriting has been proposed initially in [29], targeted for Reduced Boolean Circuits (RBC); it was later implemented and improved for ABC in [30].

2.3.4 Retiming

Retiming a sequential circuit is a standard technique used in sequential synthesis, aiming at the relocation of the registers in the circuit in order to optimize some of the circuit characteristics. Retiming can either targets the minimization of the delay in the circuit, or the minimization of the number of registers given a delay constraint, or the unconstrained minimization of the number of registers in the circuit. It does so while keeping the output functionality of the circuit intact [31]

2.3.5 Property Directed Reachability (Pdr)

The Pdr algorithm aims at proving that no violating state is reachable from the initial state of a given AIG network. It maintains a trace representing a list of over-approximations of the states reachable from the initial state, along with a set of *proof-obligations*, which can be a set of bad states or a set of states from which a bad state is reachable. Given the trace and the set of obligations, the Pdr algorithm manipulates them and keeps on adding facts to the trace until either an inductive invariant is reached and the property is proved, or a counter example is found (a bad is state

is proven to be reachable). The algorithm was originally developed by Aaron Bradley in [36, 37] and was later improved by Een et. al in [34].

2.3.6 Temporal Induction

Temporal induction carries an inductive proof of the property over the time steps of a sequential circuit. Similar to a standard inductive proof, it consists of a base case and an inductive hypothesis. These steps are typically expressed as SAT problems to be solved by traditional SAT solvers. k -step induction strengthens simple temporal inductive proofs by assuming that the property holds for the first k time steps (states), i.e. a longer base case needs to be proven [32]. Since the target is to prove unsatisfiability (proving that the negation of the property is unsatisfiable), if the base case is satisfiable, a counter-example is returned. Otherwise, the induction step is checked by assuming that the property holds for all the states except the last one (the $(k + 1)$ 'th state) [38].

2.3.7 Interpolation

Given an unsatisfiable formula $A \wedge B$, an interpolant I is a formula such that $A \implies I$, $I \wedge B$ is unsatisfiable and I contains only common variables to A and B . Given a system M , a property p and a bound k , interpolation based verification starts by attempting bounded model checking (BMC) with the bound k . If a counter-example is found, the algorithm returns. Otherwise, it partitions the problem into a prefix pre and a suffix suf , such that the problem is the conjunction of the two. Then the interpolant I of pre and suf is computed, it represents an over-approximation of the set of states reachable in one step from the initial state of the algorithm. If I contains no new states, a fixpoint is reached and the property is proved. Otherwise, the algorithm reiterates and replaces the initial states with new states added by I [33].

Table 2.1: Example ABC techniques

Technique	Description	ABC command
Balancing	Balancing reduces the number of AIG levels by applying associativity transformations [4]	<code>balance</code>
Structural Register Sweep (SRS)	SRS reduces the number of registers in the circuit by eliminating stuck-at-constant registers [28]	<code>scl -1</code>
Signal Correspondence (Scorr)	Scorr computes a set of classes of sequentially-equivalent nodes using k -step induction [28]	<code>ssweep</code>
Rewriting	AIG rewriting iteratively selects and replaces rooted AIG subgraphs with smaller pre-computed subgraphs in order to reduce the size of the AIG [29]	<code>rewrite</code>
Refactoring	AIG refactoring is a variation of AIG rewriting that uses a heuristic algorithm to compute one large cut for each AIG node, then replaces the structure of these cuts with a factored form if an improvement is observable [30]	<code>refactor</code>
Retiming	Retiming aims at manipulating registers over combinational nodes in a given logic network, while maintaining the output functionality and logic structure [31]	<code>retime</code>
Temporal Induction	Temporal induction uses SAT solvers to carry simple and k -step induction proofs over the time steps of the sequential circuit [32]	<code>ind</code>
Interpolation	Interpolation-based algorithms aim at finding interpolants in order to derive an over-approximation of the reachable states of the AIG network with respect to the property [33]	<code>int</code>
Property Directed Reachability (Pdr)	Pdr tries to prove that there is no transition from an initial state of the AIG to a bad state [34]	<code>pdr</code>

Chapter 3

$\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$: Imperative Programs to AIG

$\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ targets the verification of imperative programs annotated with precondition and postcondition specifications. An imperative program is a sequence of instructions that describes in full details the steps that the execution unit must take to accurately implement the required functionality. In order to verify that a program accurately implements its functionality, the developer provides a set of formal specifications in the form of a precondition-postcondition pair. A precondition is a FOL formula over the the program’s inputs that specifies which combination of inputs are acceptable; i.e. under which inputs the program is expected to work. A postcondition is a FOL formula over the program’s inputs and outputs that defines the program’s expected output. The postcondition relates the program’s outputs to its inputs [39].

Given a program \mathcal{S} , a precondition and postcondition pair $(\mathcal{P}, \mathcal{Q})$, and a bound b on the domain of \mathcal{S} and its variables, $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ checks whether \mathcal{S} satisfies its specifications ($\mathcal{S} \models (\mathcal{P}, \mathcal{Q})|_b$); i.e. when the bounded inputs of \mathcal{S} satisfy \mathcal{P} , the outputs of \mathcal{S} must necessarily satisfy \mathcal{Q} . $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ accepts programs written an imperative language, *Tiny*, a subset of C++ extended with support for FOL and a block synchronization construct, the `do_together` block. The tool then translates the problem ($\mathcal{S} \models (\mathcal{P}, \mathcal{Q})|_b$) into an equisatisfiable AIG using a program counter encoding. The generated AIG has a single output o that is set to 1 iff \mathcal{S} violates its specifications. The tool then uses the ABC sequential AIG model checker [4] to check that o is never set to 1. If the check is successful, \mathcal{S} satisfies its specifications. Otherwise, $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ returns the violating trace (i.e. the counterexample) to the user for debugging of \mathcal{S} .

The Array search program in Figure 3.1(a) takes as input an array a , a start index s , an end index e , a data value d , and the number of elements in the array n . It is annotated with a specification consisting of a precondition and a postcondition. The precondition states that the start s and end e indices are within array bounds and that the array size n is within the bound on array sizes. The postcondition states that if rv is valid between s and e inclusive, then $a[rv]$ must be equal to d , otherwise, rv must be invalid (-1) and all entries in a between s and e inclusive are not equal to d .

Figure 3.1(b) shows an equivalent encoding of the array search program using a program counter execution model. The equivalent program introduces Boolean

(a) Array search program	(b) Program with program counter
<pre> 1 int ArraySearch(int [] a, int d, int s, int e, int n) { 2 @pre as { 0 <= s && s <= e && e < n && n <= MAXSIZE; } 3 int i = s; // pc = pc+1 4 while(i <= e) { // pc = (i <= e) ? 5 : 10; 5 if (a[i] == d) { // pc = (a[i] == d) ? 6 : 8; 6 break;} // pc = 10; 7 else { 8 i = i+1;} // pc = 4; 9 } 10 return i; // pc = 11; rv = i; 11 @post as { 12 ((rv >= s && rv <= e) -> a[rv] == d) xor 13 (rv == -1 -> forall(int i:[s .. e]) { a[i] != d }) 14 } </pre>	<pre> 1 dotogether { 2 preas = 0 <= s && s <= e && e < n && n <= MAXSIZE; 3 //initial values 4 pc = 0; notdone = true; postas = true; } 5 while (notdone) { dotogether { // next state functions 6 i = (pc == 3) ? s : (pc == 8) ? i+1 : i; 7 notdone = (pc == 11) ? false : true; 8 rv = (pc == 10) ? i : rv; 9 pc = (pc == 0) ? 3: (pc == 3) ? 4 : 10 (pc == 4) ? ((i <= e) ? 5 : 10) : 11 (pc == 5) ? (a[i] == d) ? 6 : 8) : 12 (pc == 6) ? 10 : 13 (pc == 8) ? 4 : 14 (pc == 10) ? 11 : pc; } 15 postas = ((rv >= s && rv <= e) -> a[rv] == d) xor 16 (rv == -1 -> forall(int i:[s .. e]) { a[i] != d }); } </pre>

Figure 3.1: (a) Array search program, and (b) equivalent array search program with program counter

variables `preas`, `postas`, and `notdone` to encode the precondition, the postcondition, and the running state of the program, respectively. The equivalent program also introduces a program counter variable `pc` which encodes the control flow of the program as indicated in the comments of Figure 3.1(a). The `rv` variable denotes the return value of the original program.

The `notdone` variable is initialized to `true`, and `pc` program counter is initialized to the first executable line of the program 3. Once `pc` reaches the last executable line of the program 13, the program terminates and thus `notdone` becomes `false`. Assignment statements are grouped by target variables, and encoded into conditional assignment statements that depend on the value of `pc`. For example, the iterator `i` is assigned to `s` when `pc` is 3, incremented when `pc` is 8, and remains the same otherwise.

The program in Figure 3.1(b) is semantically equivalent to the original program in Figure 3.1(a). Furthermore, the assignment statements on Lines 1 and 3 assign initial values to the target variables. The assignment statements inside the `while` loop (Lines 5 to 13) compute the next state value of each of the target variables of the program.

Our method translates the program in Figure 3.1(b) to a sequential circuit where an iteration of the `while` loop is equivalent to a single time step in the sequential circuit. The method represents each Boolean variable with one register, and each scalar variable with a finite vector (bit-vector) of registers with initial value and next state functions. The initial state functions of the vector of registers corresponding to a variable are connected to a vector of gates that represents the right hand side initial value assignment statement of the variable. For example, `pc` ranges from 0 to 11 and can be encoded using 4 registers.

Program variables that are not initialized in the code are considered input variables and the methods connects their initial value functions to free primary inputs. The method connects the next state functions of register vectors corresponding to program variables to gates that represent the right hand side of the next state assignment. The conditional, arithmetic, and Boolean operations in the right hand side expressions

are encoded as combinational logic circuits in the usual manner.

Our method takes the resulting sequential circuit, designates a gate therein that represents $preas \wedge done \rightarrow postas$ as the output gate, passes the circuit to ABC, and checks for the validity of the designated gate. The ABC solver returns a counterexample $a = [000], s = 0, e = 1, n = 3, d = 1, rv = 2$ where d is not in a , and the return value is $e + 1$, while the postcondition requires an invalid index (-1).

The provided counter example can be used to fix the program. A possible fix is to replace Line 6 with `return i;`, and Line 10 with `return -1;`. Our method takes the fixed program, transforms it into a sequential circuit, and passes it to ABC which validates the correctness of the program modulo the finite size of the variable vectors using symbolic model checking.

3.1 Limitations of translation to CNF

Existing tools such as CBMC [5] check for pointer safety, within bound array access and user defined assertions in C programs. Given a C program and a bound on the range of variables, CBMC unwinds the program’s loops and recursive functions, and unfolds the program into a Boolean (CNF) formula that asserts the specified properties. It then uses SAT methods and tools such as MiniSat [6] to check the CNF formula for counter examples.

Recent advances in SAT enabled tools like the Alloy Analyzer [40] and CBMC to check designs of real systems. However, 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 7 nodes in a tree structure with the Alloy Analyzer.

There are three limiting aspects of translating high-level programs 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 in declarative first order logic.

Disadvantage 2 The SAT solver is restricted to using optimizations, such as symmetry breaking [41] and observability don’t cares (ODC) [42], 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 [43]. 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 programs as well as to check more sophisticated specifications 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.

The work in [44] takes a declarative formula ϕ in first order logic (FOL) with transitive closure and a bound on the universe of discourse and translates it to a sequential circuit expressed in VHDL. It then passes the sequential circuit to a sequential circuit solver and decides the validity of ϕ within the bound. It scales to bounds larger than what is possible with Kodkod [45] which translates ϕ into a propositional Boolean formula in conjunctive normal form (CNF) and checks its validity with a Boolean satisfiability solver.

The work in [46] translates an imperative C program, with an assertion statement therein, and a bound on the input size, into a sequential circuit expressed in VHDL. It then passes the sequential circuit to a sequential circuit solver and decides the validity of the assertion within the bound. It scales to bounds larger than what is possible with CBMC [47] which translates the program with a bound on the input size and the number of loop iterations into a propositional Boolean formula in conjunctive normal form (CNF) and checks for correctness using a Boolean satisfiability solver.

Our method extends the work in [44, 46] in that

- it supports function calls including recursion, and requires a bound on recursion depth only if the recursive function uses local variables,
- it enables a termination guarantee check within a bound on execution time, it then uses the execution time bound with bounded model checking to decide correctness,
- it directly translates the program into bit level representation using *and inverted graphs* (AIG) instead of the VHDL representation that requires a VHDL compiler to be translated into bit level,
- it uses ABC [4], an open source sequential circuit solver, instead of SixthSense [48] an IBM internal sequential circuit solver,
- and it is an open source tool available online ¹.

3.2 The *Tiny* imperative language

The grammar in Figure 3.2 describes *Tiny*, $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$'s imperative input language. It is composed of a subset of the C++ programming language, extended with first order logic support and some special constructs. A **program** is a list of declarations and statements. Variables can be declared to be of two kinds: (1) *wires* and (2) *registers*. Wires are non-memory elements used to monitor the values of different variables or terms in the program, at every instance in the program's execution. Once assigned to a term, wires will reflect the value of the term at every point in the program. On the contrary, register variables are memory elements that only change value once they are the target of an assignment statement executed at a specific program point. The value

¹<http://webfea.fea.aub.edu.lb/fadi/dkwk/doku.php?id=sa>

of a register variable is memorized between two different assignment statements. In what follows, we refer by wires to wire variables, and by variables to register variables.

A variable can either be single or an array. Arrays can be declared to have a constant predefined size, or can be left free to have the maximum number of elements to be determined by $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$'s runtime engine. Similarly, wires can either be singular or arrays. $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ currently support one dimensional and two dimensional arrays. Given a two dimensional array a of size n by m where n and m are constants, the tool transforms a into a single dimensional array a' of size $n \times m$, and translates all array accesses $a[i][j]$ into accesses of the form $a'[i \times (n \times m) + j]$.

Statements can be assignment statements, control statements or synchronization statements. Assignment statements modify the state of program by assigning new values to select program variables. Control statements are used to modify the control state of the program by selecting the next program location, possibly begin dictated by the value of a certain term. $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ supports the **if-then-else** selection control statement and the **while** loop control statement. Synchronization statements start with the **dotogether** modifier, and are used to enforce the execution of a list of data-independent assignment or selection statements at the same program point (equivalently, at the same clock cycle).

Additionally, $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ provides support for *function definitions* and *function calls*. Defining and calling a function is done in the same manner as in a regular C++ program, with the exception that **return** statements must always be present at the end of a function's body. $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ also allows the declaration of recursive functions. Expressions in *Tiny* extend the definition of terms presented in Definition 1 of Chapter 2 with the addition of allowing terms to also be function calls, as depicted by the notation `term_with_function_call` on line 31 of Figure 3.2.

$\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ extends the subset of C++ above with support for FOL specifications, written in the form of *pre-condition*, *post-condition* pairs. FOL expressions are Boolean expressions that can be Boolean terms or function calls, or *quantified expressions*. A quantified expression is either universally (**forall**) or existentially (**exists**) quantified.

3.3 Translation to AIG

Given an imperative program \mathcal{S} written in the *Tiny* language, and annotated with a FOL precondition postcondition specification pair $(\mathcal{P}, \mathcal{Q})$, $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ transforms the tuple $(\mathcal{S}, (\mathcal{P}, \mathcal{Q}))$ into an *CAIG* program \mathcal{S}' . The tool then synthesizes \mathcal{S}' into an equisatisfiable AIG C having a single output representing the formula $\neg(\mathcal{P} \wedge \mathcal{S} \implies \mathcal{Q})$. The tool then uses the ABC AIG solver to verify the validity of $\neg o$ and thus prove that \mathcal{S} satisfies its specification pair $(\mathcal{P}, \mathcal{Q})$ if successful. If the validity check fails, the tool returns a counterexample to be used by the developer for debugging.

3.3.1 Preprocessing

$\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ first starts by transforming \mathcal{S} into an intermediate program $\mathcal{S}' = \rho(\mathcal{S})$ where ρ is a code transformation function that simplifies function calls, recursive functions


```

1 program: block+
2 block: (declaration | statement)
3 statement: assignment_statement | conditional_statement |
4             loop_statement | sync_statement
5
6 // declarations
7 declaration: variable_declaration | function_declaration | property_declaration
8 variable_declaration: modifier? type id ('[' num ']') ('[' num ']')?
9                       ('=' term)?
10 function_declaration: type id '(' argument_list? ')
11                      '{' block* return_statement '}'
12
13 // statements
14 assignment_statement: target '=' expression ';'
15 sync_statement: '@dotogether' '{' (assignment_statement | conditional_statement)+ '}'
16 conditional_statement: 'if' '(' expression ')' '{' block '}'
17                      'else' '{' block+ '}'
18 loop_statement: 'while' '(' expression ')'
19               '{' block+ '}'
20 return_statement: 'return' expression ';'
21
22 // properties
23 property_declaration: precondition | postcondition
24 precondition: '@pre' id '{' property+ '}'
25 postcondition: '@post' id '{' property+ '}'
26 property: expression | quantified_property
27 quantified_property: ('forall'|'exists') '(' range ')'
28                   '{' property+ ';' '}'
29
30 // expressions
31 expression: term_with_function_call
32 function call: id '(' call_arguments? ')'
33 call_argument: id (',' id)*
34
35 argument_list: variable_declaration (',' variable_declaration)*
36 modifier: 'wire' | 'const'
37 specifier: 'int' | 'bool'
38 target: id | target '[' expression ']'
39 range: id '[' expression '...' expression ']'

```

Figure 3.2: The *Tiny* imperative language

and properties.

Function calls. ρ does not inline functions; instead, it uses a program counter mechanism to avoid inlining and reuse the same code and thus the same AIG after synthesis. Key to that is the transformation of function class as follows. Let $f(args_f)$ be a call to function declaration $f_d(args_{f_d})$ where $args_f$ is a list of expressions passed as arguments to f , and $args_{f_d}$ is the list of arguments declared in f_d . Let $ret(f_d)$ be

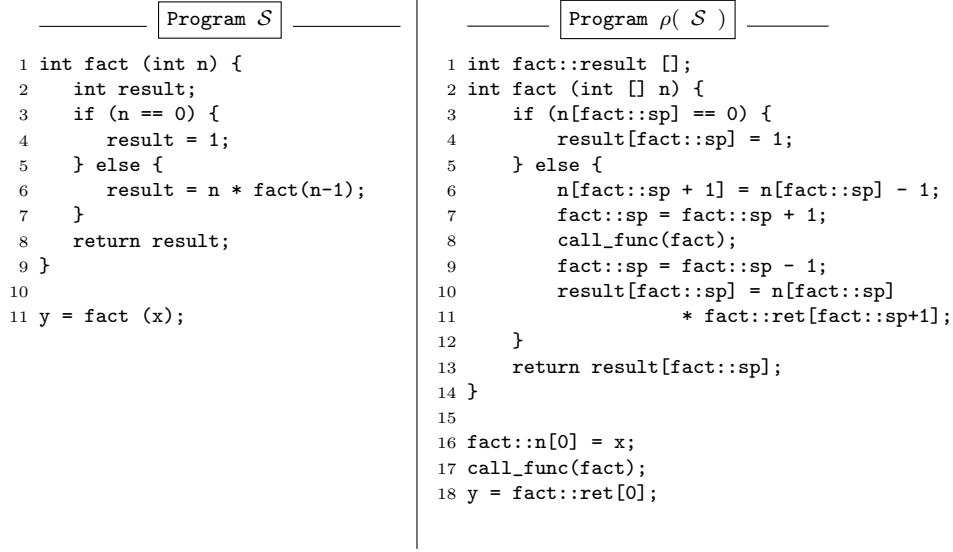


Figure 3.3: Transformation function ρ on function calls

the return variable of f_d , and $args_f(i)$ and $args_{f_d}(i)$ be the i 'th argument passed to f and declared in f_d , respectively, for $0 \leq i \leq |args_f|$.

Consider the assignment statement $s = (target := f(args_f))$ where $target$ is the target of the assignment as defined in line 38 of Figure 3.2. The transformation function $\rho(s)$ is defined as follows:

$$\rho(s) = \left\{ \begin{array}{l} \left(;_{i=0}^{|args_f|} (args_{f_d}(i) := args_f(i)) \right); \\ call_func(f_d) ; (target := ret(f_d)) \end{array} \right. \quad (3.1)$$

The $;$ operator represents an ordering of the statements, where $s_1; s_2$ means that s_1 executes before (or possibly at the same time as) s_2 . Intuitively, ρ copies the arguments passed to the call f onto the arguments declared in the function declaration f_d . It then adds the `call_func(f_d)` statement, a special statement that redirects the control of the program to the starting point of f_d 's body. It finally assigns that $target$ of s to the return variable of the function f_d .

Recursive functions. Let $f_d(args_{f_d})$ be a recursive function with arguments $args_{f_d}$, and let $f(args_f)$ be a *recursive call* to f_d , i.e. a call to f_d from inside the body of f_d . Using ρ , $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ emulates recursion by (1) adding a stack pointer variable sp_{f_d} that maintains the recursive depth of the current function call, and (2) increasing the dimensionality of all arguments and local variable of f_d by 1. In other words, variables become arrays and arrays become two dimensional arrays. Subsequently, all assignments and references to arguments or local variables of f_d are replaced by array access terms indexed by the current value of sp_{f_d} . The recursion depth (i.e. sp_{f_d}) is incremented before each *recursive call* f and decremented once it returns.

Consider the assignment statement $s = (target := f(args_f))$ where $target$ is the target of the assignment as defined in line 38 of Figure 3.2, and f is a recursive call to f_d . The transformation function $\rho(s)$ is then defined as follows:

$$\rho(s) = \begin{cases} \left(\begin{array}{l} \left(\underset{i=0}{\overset{|args_f|}{\cdot}} (args_{f_d}(i)[sp_{f_d} + 1] := args_f(i)) \right); \\ (sp_{f_d} := sp_{f_d} + 1); call_func(f_d); (sp_{f_d} := sp_{f_d} - 1) \\ (target := ret(f_d)[sp_{f_d} + 1]); \end{array} \right); \end{cases} \quad (3.2)$$

Note that arguments of f_d to be assigned are indexed by the future value of the recursion stack depth pointer (i.e. $sp_{f_d} + 1$) before the the pointer is incremented. The same is applied to the return variable of f_d after the pointer has been decremented.

Additionally, consider the assignment statement $s = (target := f(args_f))$ where f is a non-recursive call to f_d , sp_{f_d} is guaranteed to have a value of 0 and the transformation function $\rho(s)$ is defined as:

$$\rho(s) = \begin{cases} \left(\begin{array}{l} \left(\underset{i=0}{\overset{|args_f|}{\cdot}} (args_{f_d}(i)[0] := args_f(i)) \right); \\ call_func(f_d); (target := ret(f_d)[0]); \end{array} \right); \end{cases} \quad (3.3)$$

Figure 3.3 shows an example of applying the transformation function ρ onto a program containing a recursive function call **fact**(**n**) that computes the factorial of an integer **n**. The argument **n** of **fact** and the local variable **result** are transformed into the arrays **fact::n** and **fact::result**. **fact::sp** is the recursive stack pointer variable added by ρ for the function **fact**. Local references to **n** and **result** are replaced by the sequence access terms **fact::n[fact::sp]** and **result[fact::sp]**.

Lines 6-11 of $\rho(\mathcal{S})$ in Figure 3.3 show the result of applying ρ on the recursive function call **fact**(**n-1**) on line 6 of the program \mathcal{S} . The next value of the argument n is assigned to the current value of n decremented by 1, as shown in the statement **fact::n[fact::sp + 1] = fact::n[fact::sp] - 1**. The stack pointer variable is then incremented before adding the function call statement **call_func(fact)** that gives the control to the body of the function **fact**. Once the function call returns, the stack pointer variable is decremented, and any reference to the return variable of **fact** is replaced by an array access term to the next value of the stack pointer (**fact::sp + 1**). Lines 16-18 of \mathcal{S}' shows the result of applying ρ to a non-recursive call to **fact**, in which the arguments and return variable of **fact** are replaced by array access terms indexed by 0.

Quantifiers. Consider the assignment statement $s = (target := Q(i : [t_1 \dots t_2])\{\mathcal{B}\})$ where $target$ is the target variable of the assignment, Q is either ‘forall’ or ‘exists’, i is a quantified variable, t_1 and t_2 are terms representing the range of i , and \mathcal{B} is a Boolean FOL formula. $\rho(s)$ is defined as follows:

$$\rho(s) = \begin{cases} (Q_r := true); (i := t_1); \\ while(i \leq t_2) \{ \rho(Q_r := Q_r (\&\& \text{ or } ||) \mathcal{B}); (i := i + 1) \}; \\ (target := Q_r) \end{cases} \quad (3.4)$$

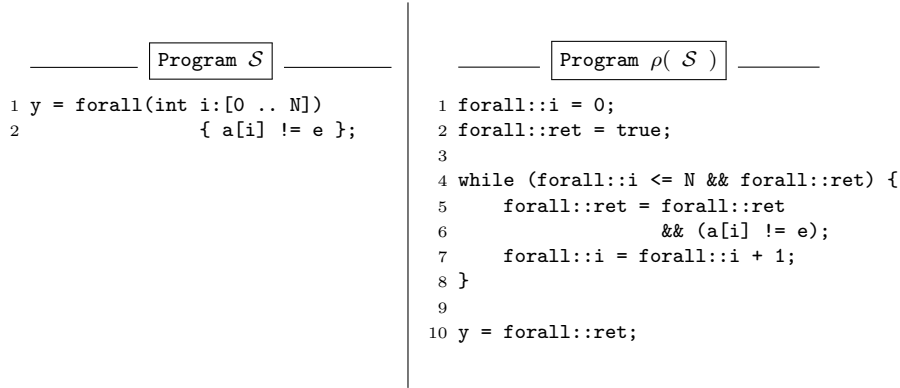


Figure 3.4: Transformation function ρ on universal quantifier

For the quantified expression $(Q(i : [t_1 \dots t_2])\{\mathcal{B}\})$, ρ adds a Boolean return variable Q_r that is initialized to *true* and holds the value of the expression. Then ρ transforms the expression into a while loop that iterates over all possible values of i , and assigns the return variable Q_r to its current value conjuncted or disjuncted with the value of \mathcal{B} . The type of the operation performed is determined by the nature of the quantifier; conjunction for the universal quantifier (*forall*) and disjunction for the existential quantifier (*exists*). ρ finally adds the actual assignment statement of *target* as $target := Q_r$. Note that ρ is also applied to the update assignment to the variable Q_r in order to handle the cases where \mathcal{B} is also quantified or it contains a function call.

Figure 3.4 shows an example of applying the transformation function ρ on a program \mathcal{S} containing an assignment of the variable y to a universal quantifier. Note that we added the expression `forall::ret` to the condition of the `while` loop in order to allow for early exit of the loop. Additionally, in the case the bounds on the quantified variable are constants, we optimize the transformation by unrolling the loop into a single large Boolean expression.

Pre/Post conditions. Consider the *Tiny* precondition and postcondition declarations $@pre \mathcal{P} \{\mathcal{B}\}$ and $@post \mathcal{Q} \{\mathcal{F}\}$ where \mathcal{B} and \mathcal{F} are Boolean expressions. For brevity, we refer by \mathcal{P} and \mathcal{Q} to the precondition and postcondition declarations, respectively. The transformation function ρ is defined as follows:

$$\rho(\mathcal{P}) = \rho(x_{\mathcal{P}} := \mathcal{B}) \tag{3.5}$$

$$\rho(\mathcal{Q}) = \rho(x_{\mathcal{Q}} := \mathcal{F}); \text{assert}(x_{\mathcal{P}} \implies x_{\mathcal{Q}}) \tag{3.6}$$

ρ creates for each declaration a Boolean variable that is used to hold its value across the program's execution. ρ then replaces the precondition declaration \mathcal{P} by a statement that assign its created variable $x_{\mathcal{P}}$ to its declared FOL formula \mathcal{B} . Similarly, ρ replaces the postcondition declaration \mathcal{Q} by the statement $x_{\mathcal{Q}} := \mathcal{F}$. In order to resolve quantification or function calls in \mathcal{B} and \mathcal{F} , we also apply ρ to the created statements.

Additionally, ρ adds an assertion statement $\text{assert}(x_{\mathcal{P}} \implies x_{\mathcal{Q}})$ after the postcondition assignment statement. This assertion statement will not be evaluated,

instead it used the help $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ determine the outputs to pass to the ABC AIG solver for verification.

All statements and expressions in \mathcal{S} that are not mentioned in this Section are kept unchanged by the transformation function ρ .

3.4 Transformation to \mathcal{CAIG}

After preprocessing, $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ transforms the program $\mathcal{S}' = \rho(\mathcal{S})$ into an equisatisfiable \mathcal{CAIG} program \mathcal{C} . Synthesizing an AIG from \mathcal{C} is then a direct translation of variables into bit registers and building their next state function according to the `next` block definition in \mathcal{C} . $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ uses *program counter semantics* to translate \mathcal{S}' into \mathcal{C} . The program counter is used to ensure the correct sequencing of assignments in the program; i.e. to provide the concept of time.

Our method assigns a unique *label* for each statement in the program \mathcal{S}' . Let s be any statement in \mathcal{S}' , and let s_a, s_i, s_w , and f_d be an assignment statement, a conditional statement, a loop statement and a function declaration in \mathcal{S}' , respectively. Additionally, let e be an array access expression, and let $call(f_d)$ be an added `call_func` statement that calls the function declaration f_d . Our method defines the following functions:

- `label(s)`: The unique label identifying the statement s
- `next(s)`: The label of the statement that directly follows s in the program order.
- `condition(s_i)`: The Boolean condition of the selection statement s_i .
- `then(s_i)`: The label of the first statement in the *then* code block of s_i .
- `else(s_i)`: The label of the first statement in the *else* code block of s_i .
- `condition(s_w)`: The Boolean condition of the loop statement s_w .
- `body(s_w)`: The label of the first statement in the body of the loop s_w .
- `last(s_w)`: The last statement in the body of the loop s_w .
- `target(s_a)`: The target variable of the assignment s_a .
- `expression(s_a)`: The expression to be assigned to the target of s_a .
- `body(f_d)`: The label of the first statement in the body of the function f_d .
- `return(call(f_d))`: The label of the statement to which the function call statement should return after calling f_d .
- `base(e)`: The array which the expression e is indexing.
- `index(e)`: The index at which the base of the expression e is indexed.

Algorithm 1 shows the procedure used in $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ to translate a preprocessed *Tiny* program \mathcal{S}' with a set of variables V and an initial entry statement, into an *CAIG* program \mathcal{C} , to be used for the synthesis of the equisatisfiable AIG. Intuitively, the wire declarations, variable definitions and wire definitions are directly moved from \mathcal{S}' to their corresponding blocks in \mathcal{C} . Additionally, in order to model time, $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ adds a new scalar variable to the program \mathcal{S}' , the *program counter* pc . This variable is key to avoid inlining functions and unrolling loops. It is used to ensure the proper sequencing of the statements of \mathcal{S}' ; the current value of the pc variable defines which statement is to be execute. For example, reassigning the value of pc to the starting point of a loop's body allows $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ to make use of the same code block (thus the same AIG) to execute a single loop infinitely many times.

The algorithm starts by building the initialization list of \mathcal{C} . It assigns the program counter variable pc to the label of the entry statement of \mathcal{S}' . Then for each variable $v \in V$ that is used before being assigned (i.e. used with an undefined value), it assigns v to a set of free inputs (AIG primary inputs) who's values are to be set by the ABC AIG solver. Note that since variables can be initialized at different points in the program (and not necessarily at the starting point of the algorithm), we delay their initialization step into the *next* block, where these variables will get their initial value at the appropriate pc value.

Subsequently, the algorithm moves to building the *next* state block of \mathcal{C} . For each variable $v \in V$, and for each assignment statement s such that $target(s) = v$, the next state of v will be $expression(s)$ iff $pc == label(s)$. If at a given statement label, v is not being assigned, it retains its current value. For example, consider the following assignment statements for v :

```

1 (11): x = 1;
2 (12): x = 3;
3 (13): x = 5;

```

The algorithm builds the next state function of x as the following statement

```

1 x = (pc == 11)? 1 : (pc == 12)? 3 : (pc == 13)? 5 : x;

```

After traversing all of \mathcal{S}' 's variables, the algorithm then builds the next state assignment for the program counter. At a given statement s , the default behavior of the pc is to move from the current label $label(s)$ to the label of the statement that directly follows s , i.e. $next(s)$. Only control statements such as conditional statements, loop statements, function call statements and return statements are allowed to alter the default behavior of pc .

Conditional statements alter the value of the program counter based on the evaluation of their conditional expression. Consider a conditional statement s and let $e = condition(s)$. When pc is at $label(s)$, its next value is either the label of the first statement in then *then* branch of s if e evaluates to *true*, or to the first statement in the *else* branch of s otherwise. Therefore at $label(s)$, the next state of the pc is defined as $(e)? then(s) : else(s)$.

Similarly, loop statements move the program counter into their body when their conditional expression evaluates to *true*, and to their next statement once it is *false*. Consider a loop statement s and let $e = condition(s)$. At $label(s)$, the next state

of pc is defined as $(e)? body(s) : next(s)$. Note that according to the *Tiny*, the loop statement block is considered as a single statement, and thus the next statement to follow s would be the first statement outside its body, i.e. its exit point. Additionally, for the last statement in the body of the loop, its next statement is defined as the loop statement itself. This ensure that once the pc reaches that last statement in the loop, it once again evaluates its condition in order to determine whether to exit or stay in the loop.

Finally, function call statements and return statements redirect the program counter into the body of the function and to the first statement after the call, respectively. Note that for a non-recursive function, there is at most one active function call at a time, therefore it is easy to track the label at which the program counter should return to. For a recursive function, we maintain a stack of function calls and redirect the control to statement after the call that is on top of the stack.

Post-processing

After constructing the *CAIG* program \mathcal{C} , $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ employs a last post-processing step to resolve array accessing. Given an array access expression $a[i]$ where a is an array and i is an index, resolving this access expression is to transform it into an array access where the index is constant. An array access expression with a constant index is handled as a regular register variable in \mathcal{C} .

The first part of algorithm 2 shows the post processing of a reference to an array access expression e . The algorithm pushes the variable index i outside of the array access by creating a set of checks for the value of i at each possible index of the array a , and returning the appropriate array element accordingly.

The second part of algorithm 2 shows the post processing of statement s where $target(s)$ is an array access of the form $a[i]$. Since this assignment statement can be to any of the elements of the array a , as dictated by the value of i , the algorithm creates an assignment statement for each possible element of a . This ensure that only the correct element get assigned to the value of $expression(e)$ by creating a check for i at each possible index of the array.

Figure 3.5 shows an *CAIG* program \mathcal{C} that contains array access assignments and references. Post-processing transforms the assignment statement on line 14 of the program \mathcal{C} into assignment statements over all the elements of the array a , with added checks on the value of the index i . Also post-processing replaces the expression $b[k]$ in the right hand side of the assignment statements with the expression $(k == 0)? b[0] : b[1]$. This expression checks on the value of k for all possible indices to the array b and returns the corresponding element.

3.5 *CAIG* to AIG

Given a bound b on the bit width of variables, $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ synthesizes an AIG by a direct translation of a *CAIG* program \mathcal{C} into a sequential circuit. The synthesis proceeds as follows. Scalar register variables and array elements are translated into vectors of b bits registers, while Boolean variables and array elements are directly mapped onto one bit

Program \mathcal{C}	\mathcal{C} after post-processing
<pre> 1 int a [2]; 2 int b [2]; 3 int i, k; 4 @dotogether { 5 a[0] = free_inputs; 6 a[1] = free_inputs; 7 i = free_inputs; 8 k = free_inputs; 9 b[0] = free_inputs; 10 b[1] = free_inputs; 11 } 12 13 @dotogether { 14 a[i] = (pc == 11)? b[k] : a[i]; 15 } </pre>	<pre> 1 int a [2]; 2 int b [2]; 3 int i, k; 4 @dotogether { 5 a[0] = free_inputs; 6 a[1] = free_inputs; 7 i = free_inputs; 8 k = free_inputs; 9 b[0] = free_inputs; 10 b[1] = free_inputs; 11 } 12 13 @dotogether { 14 a[0] = (i == 0)? 15 ((pc == 11)? ((k == 0)? b[0] : b[1]) : a[0]) 16 : a[0]; 17 a[1] = (i == 1)? 18 ((pc == 11)? ((k == 0)? b[0] : b[1]) : a[1]) 19 : a[1]; 20 } </pre>

Figure 3.5: Post processing of a *CAIG* \mathcal{C}

registers. The `init` block of \mathcal{C} is used to determine the initial value of the instantiated registers.

The `next` block of \mathcal{C} infers the next state functions for all registers in the AIG, including the program counter. For each variable, we translate the expression of its next state function into a hierarchy of multiplexers that reflect the ternary choice expressions in \mathcal{C} 's assignment expressions.

Finally, $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ builds the primary output of the AIG as the negation of the assertion statement introduced in the pre-processing step. Given a precondition \mathcal{P} and a postcondition \mathcal{Q} , the primary output o will be $\neg(x_{\mathcal{P}} \implies x_{\mathcal{Q}})$ where $x_{\mathcal{P}}$ and $x_{\mathcal{Q}}$ are the precondition and postcondition variables introduced in the pre-processing step.

Algorithm 1 *Tiny* to *CAIG* transformation algorithm

```
1: Input: Tiny program  $\mathcal{S}$ , entry_statement
2: Output: CAIG program  $\mathcal{C}$ 
3:
4: // build the initialization list
5: init_list []
6: for all  $v \in V$  do
7:   if used_before_assigned( $v$ ) then
8:     init_list.insert (“ $v = \text{free\_inputs}$ ”)
9:   end if
10: end for
11: init_list.insert (“ $pc = \text{label}(\text{entry\_statement})$ ”)
12:
13: // build the next state list for variables
14: next_state_list []
15: for all  $v \in V$  do
16:    $\text{next}_v = “v = ”$ 
17:   for all Assignment statement  $s$  such that  $\text{target}(s) = v$  do
18:      $\text{next}_v += “(pc == \text{label}(s))? \text{expression}(s) : ”$ 
19:   end for
20:    $\text{next}_v += “v”$ 
21:   next_state_list.insert ( $\text{next}_v$ )
22: end for
23:
24: // build the next state for the program counter
25:  $n_{pc} = “pc = ”$ 
26: for all statement  $s$  do
27:   if is_conditional_statement ( $s$ ) then
28:      $n_{pc} += “(pc == \text{label}(s))? (\text{condition}(s)? \text{then}(s) : \text{else}(s)) : ”$ 
29:   else if is_loop_statement ( $s$ ) then
30:      $n_{pc} += “(pc == \text{label}(s))? (\text{condition}(s)? \text{body}(s) : \text{next}(s)) : ”$ 
31:   else if is_function_call( $s$ ) then
32:      $f_d := \text{function\_declaration\_of}(s)$ 
33:      $n_{pc} += “(pc == \text{label}(s))? (\text{body}(f_d)) : ”$ 
34:   else if is_return_statement ( $s$ ) then
35:      $f_d := \text{function\_declaration\_of}(s)$ 
36:      $n_{pc} += “(pc == \text{label}(s))? (\text{return}(\text{call}(f_d))) : ”$ 
37:   else
38:      $n_{pc} += “(pc == \text{label}(s))? (\text{next}(s)) : ”$ 
39:   end if
40: end for
41: next_state_list.insert ( $n_{pc}$ )
```

Algorithm 2 Array access resolving algorithm

```
1: // array access on right hand side
2: Input: Array access expression  $e$ 
3: Output: Array access expressions  $e'$  with constant indexing
4:  $a := base(e)$ 
5:  $i := index(e)$ 
6:  $N := size(a) - 1$ 
7: for  $j \in 0 \dots N - 1$  do
8:    $e' += (i == j)? a[j] :$ 
9: end for
10:  $e' += a[N]$ 
11:
12: // array access on left hand side
13: Input: Statement of the form  $a[i] = e$ ,  $e$  is an expression
14: Output: A list of statements where all array accesses have a constant index
15: List_of_statements []
16:  $N := size(a) - 1$ 
17: for  $j \in 0 \dots N - 1$  do
18:   List_of_statements.insert ( $a[j] = (i == j)? e : a[j]$ )
19: end for
```

Chapter 4

$BIP\{I\}$: BIP to AIG

4.1 BIP - Behavior Interaction Priority

We recall the necessary concepts of the BIP framework [8]. BIP allows to construct systems by superposing three layers of modeling: Behavior, Interaction, and Priority. The *behavior* layer consists of a set of atomic components represented by transition systems. The *interaction* layer models the collaboration between components. Interactions are described using sets of ports. The *priority* layer is used to specify scheduling policies applied to the interaction layer, given by a strict partial order on interactions.

Figure 4.1 shows a traffic light controller system modeled in BIP. It is composed of two atomic components, **timer** and **light**. The timer counts the amount of time for which the light must stay in a specific state (i.e. a specific color of the light). The light component determines the color of the traffic light. Additionally, it informs the timer about the amount of time to spend in each location through a data transfer on the interaction between the two components.

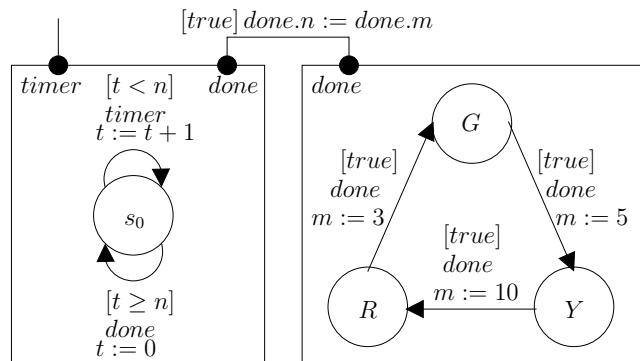


Figure 4.1: Traffic light controller BIP system

4.1.1 Component-based Construction

BIP offers primitives and constructs for modeling and composing complex behaviors from atomic components. Atomic components are Labeled Transition Systems (LTS) extended with C functions and data. Transitions are labeled with sets of communication ports. Composite components are obtained from atomic components by specifying connectors and priorities.

Atomic Components.

An atomic component is endowed with a finite set of local variables X taking values in a domain Data. Atomic components synchronize and exchange data with each others through *ports*.

Definition 8 (Port). *A port $p[x_p]$, where $x_p \subseteq X$, is defined by a port identifier p and some data variables in a set x_p (referred to as the support set). We denote by $p.X$ the set of variables assigned to the port p , that is, x_p .*

Definition 9 (Atomic component). *An atomic component B is defined as a tuple $(P, L, T, X, \{g_\tau\}_{\tau \in T}, \{f_\tau\}_{\tau \in T})$, where:*

- (P, L, T) is an LTS over a set of ports P . L is a set of control locations and $T \subseteq L \times P \times L$ is a set of transitions.
- X is a set of variables.
- For each transition $\tau \in T$:
 - g_τ is a Boolean condition over X : the guard of τ ,
 - $f_\tau = \{(x, f^x(X)) \mid x \in X\}$ where $(x, f^x(X)) \in f_\tau$ expresses the assignment statement $x := f^x(X)$ updating x with the value of the expression $f^x(X)$.

For $\tau = (l, p, l') \in T$ a transition of the internal LTS, l (resp. l') is referred to as the source (resp. destination) location and p is a port through which an interaction with another component can take place. Moreover, a transition $\tau = (l, p, l') \in T$ in the internal LTS involves a transition in the atomic component of the form $(l, p, g_\tau, f_\tau, l')$ which can be executed only if the guard g_τ evaluates to *true*, and f_τ is a computation step: a set of assignments to local variables in X .

In the sequel we use the dot notation. Given a transition $\tau = (l, p, g_\tau, f_\tau, l')$, $\tau.src$, $\tau.port$, $\tau.guard$, $\tau.func$, and $\tau.dest$ denote l , p , g_τ , f_τ , and l' , respectively. Also, the set of variables used in a transition is defined as $\varphi(f_\tau) = \{x \in X \mid x := f^x(X) \in f_\tau\}$. Given an atomic component B , $B.P$ denotes the set of ports of the atomic component B , $B.L$ denotes its set of locations, etc.

Given a set X of variables, we denote by \mathbf{X} the set of valuations defined on X . Formally, $\mathbf{X} = \{\sigma : X \rightarrow \text{Data}\}$, where Data is the set of all values possibly taken by variables in X .

Semantics of Atomic Components. The semantics of an atomic component is an LTS over configurations and ports, formally defined as follows:

Definition 10 (Semantics of Atomic Components). *The semantics of the atomic component $B = (P, L, T, X, \{g_\tau\}_{\tau \in T}, \{f_\tau\}_{\tau \in T})$ is defined as the labeled transition system $S_B = (Q_B, P_B, T_B)$, where:*

- $Q_B = L \times \mathbf{X}$, where \mathbf{X} denotes the set of valuations on X ,
- $P_B = P \times \mathbf{X}$ denotes the set of labels, that is, ports augmented with valuations of variables,
- T_B is the set of transitions defined as follows. $T_B = \{((l', v'), p(v_p), (l, v)) \in Q_B \times P_B \times Q_B \mid \exists \tau = (l', p[x_p], l) \in T : g_\tau(v') \wedge v = f_\tau(v'/v_p)\}$, where v_p is a valuation of the variables of p .

A configuration is a pair $(l, v) \in Q_B$ where $l \in L$ is a control location, $v \in \mathbf{X}$ is a valuation of the variables in X . The evolution of configurations $(l', v') \xrightarrow{p(v_p)} (l, v)$, where v_p is a valuation of the variables attached to the port p , is possible if there exists a transition $(l', p[x_p], g_\tau, f_\tau, l)$, such that $g_\tau(v') = \text{true}$. As a result, the valuation v' of variables is modified to $v = f_\tau(v'/v_p)$.

Creating composite components.

Assuming some available atomic components B_1, \dots, B_n , we show how to connect the components in the set $\{B_i\}_{i \in I}$ with $I \subseteq [1, n]$ using an *interaction*. An interaction a is used to specify the sets of ports that have to be jointly executed.

Definition 11 (Interaction). *An interaction a is a tuple (P_a, G_a, F_a) , where:*

- $P_a \subseteq \cup_{i=1}^n B_i.P$ is a nonempty set of ports that contains at most one port of every component, that is, $\forall i : 1 \leq i \leq n : |B_i.P \cap P_a| \leq 1$. We denote by $X_a = \cup_{p \in P_a} p.X$ the set of variables available to a ,
- $G_a : \mathbf{X}_a \rightarrow \{\text{true}, \text{false}\}$ is a guard,
- $F_a : \mathbf{X}_a \rightarrow \mathbf{X}_a$ is an update function.

P_a is the set of connected ports called the support set of a . For each $i \in I$, x_i is a set of variables associated with the port p_i .

Definition 12 (Composite Component). *A composite component is defined from a set of available atomic components $\{B_i\}_{i \in I}$ and a set of interactions $\gamma = \{a_j\}_{j \in J}$. The connection of the components in $\{B_i\}_{i \in I}$ using the set γ of connectors is denoted by $\gamma(\{B_i\}_{i \in I})$.*

Definition 13 (Semantics of Composite Components). *A state q of a composite component $\gamma(\{B_1, \dots, B_n\})$, where γ connects the B_i 's for $i \in [1, n]$, is an n -tuple $q = (q_1, \dots, q_n)$ where $q_i = (l_i, v_i)$ is a state of B_i . Thus, the semantics of $\gamma(\{B_1, \dots, B_n\})$ is precisely defined as the labeled transition system $S = (Q, \gamma, \longrightarrow)$, where:*

- $Q = B_1.Q \times \dots \times B_n.Q$,
- \longrightarrow is the least set of transitions satisfying the following rule:

$$\frac{a = (\{p_i\}_{i \in I}, G_a, F_a) \in \gamma \quad G_a(\{v_{p_i}\}_{i \in I}) \quad \forall i \in I : q_i \xrightarrow{p_i(v_i)} q'_i \wedge v_i = F_a^i(\{v_{p_i}\}_{i \in I}) \quad \forall i \notin I : q_i = q'_i}{(q_1, \dots, q_n) \xrightarrow{a} (q'_1, \dots, q'_n)}$$

where v_{p_i} denotes the valuation of the variables attached to the port p_i and F_a^i is the partial function derived from F_a restricted to the variables associated with p_i .

The meaning of the above rule is the following: if there exists an interaction a such that all its ports are enabled in the current state and its guard evaluates to *true*, then the interaction can be fired. When a is fired, all involved components evolve according to the interaction and uninvolved components remain in the same state.

Notice that several distinct interactions can be enabled at the same time, thus introducing non-determinism in the product behavior. One can add priorities to reduce non-determinism. In this case, one of the interactions with the highest priority is chosen non-deterministically.¹

Definition 14 (Priority). Let $S = (Q, \gamma, \longrightarrow)$ be the behavior of the composite component $\gamma(\{B_1, \dots, B_n\})$. A priority model π is a strict partial order on the set of interactions A . Given a priority model π , we abbreviate $(a, a') \in \pi$ by $a \prec_\pi a'$ or $a \prec a'$ when clear from the context. Adding the priority model π over $\gamma(\{B_1, \dots, B_n\})$ defines a new composite component $\pi(\gamma(\{B_1, \dots, B_n\}))$ noted $\pi(S)$ and whose behavior is defined by $(Q, \gamma, \longrightarrow_\pi)$, where \longrightarrow_π is the least set of transitions satisfying the following rule:

$$\frac{q \xrightarrow{a} q' \quad \neg(\exists a' \in A, \exists q'' \in Q : a \prec a' \wedge q \xrightarrow{a'} q'')}{q \xrightarrow{a}_\pi q'}$$

An interaction a is enabled in $\pi(S)$ whenever a is enabled in S and a is maximal according to π among the active interactions in S .

Finally, we consider systems defined as a parallel composition of components together with an initial state.

Definition 15 (System). A BIP system \mathcal{S} is a tuple (B, Init, v) where B is a composite component, $\text{Init} \in B_1.L \times \dots \times B_n.L$ is the initial state of B , and $v \in \mathbf{X}^{\text{Init}}$ where $X^{\text{Init}} \subseteq \cup_{i=1}^n B_i.X$.

4.2 BIP to *CALG*

Given a BIP system $\mathcal{S} = (B, \text{Init}, v)$, *BIP*{ \mathcal{I} } translates \mathcal{S} into an *CALG* program \mathcal{C} with its own customized execution engine. Listing 4.1 shows the algorithm used

¹The BIP engine implementing this semantics chooses one interaction at random, when faced with several enabled interactions.

by $\mathcal{BIP}\{\mathcal{I}\}$ to implement this translation. The algorithm is divided into 4 steps: (1) wire and variable declarations, (2) wire definitions, (3) `init` block and (4) `next` block construction. The lists `decl-list`, `wiredef-list`, `init-list` and `next-list` represent the list of declarations and lists of statements used in the declaration, wire definition, `init` and `next` blocks of \mathcal{C} , respectively.

Given a port p from the system \mathcal{S} , we define the function $interaction(p) = \{a \in \gamma \mid p \in a.P\}$ where γ is the set of interactions in the composite component B of \mathcal{S} . This function returns the set of interactions in which the port p takes part. Additionally, the function $component(p) = B_i$ such that $p \in B_i.P$ for $i \in I$ defines the component to which the port p belongs. Furthermore the function $transitions(p) = \{\tau \in component(p) \mid \tau \in B.T \wedge \tau.port = p\}$ defines the set of transitions τ in the component $B = component(p)$ such that τ is labeled by p .

Intuitively, for each component B_i in B , $\mathcal{BIP}\{\mathcal{I}\}$ defined a scalar variable $B_i.l$ that represents the control location $l \in B_i.L$ at which the component B_i is currently at. Additionally, it declares a scalar variable $B_i.x_j$ for each variable $x_j \in B_i.X$. The algorithm defines the initial values for control locations and data variables based on the *Init* relation of the system \mathcal{S} . It thus builds the initialization list by adding the assignments “ $B_i.l := Init.B_i$ ” and “ $B_i.x_j := v(B_i.x_j)$ ” for $j \in [1 \dots |B_i.X|]$ and $i \in [1 \dots |I|]$, where $Init.B_i$ is the initial location from which B_i starts and $v(B_i.x_j)$ is the initial valuation of the variable $x_j \in B_i.X$.

For each atomic component B_i in B , and for every port $p \in B_i.P$, $\mathcal{BIP}\{\mathcal{I}\}$ defines two Boolean wires. (1) The *port enablement* wire ($B_i.p_j.e$) that is set to *true* iff the port $p_j \in B_i$ is *enabled*. The port p_j is enabled iff there exists a transition $\tau \in transitions(p_j)$ such that the guard of τ evaluates to *true* and B_i is in the control location $l = \tau.src$. (2) The *port selected* wire ($B_i.p_j.s$) that is set to *true* iff the port $p_j \in B_i$ is *selected*. The port p_j is selected iff there exists an interaction $a \in interactions(p_j)$ that is selected for execution by the BIP engine.

For each interaction $a_k \in \gamma$, $\mathcal{BIP}\{\mathcal{I}\}$ defines three Boolean wires. (1) The *interaction enablement* wire, $ie[k]$, that is set to *true* iff the interaction a_k is enabled. a_k is enabled iff the guard of the interaction $a_k.guard$ evaluates to *true* and all ports $p \in a_k.P$ are enabled. (2) The *interaction priority* wire, $ip[k]$, that is set to *true* iff the interaction a_k is enabled and there does not exist any interaction $a_j \in \gamma$ such that a_j is enabled and has a higher priority than a_k . (3) The *interaction selected* wire, $is[k]$, that is set to *true* iff the interaction a_k is selected for execution. Given a set of enabled interactions with equal priority, the BIP engine synthesized by $\mathcal{BIP}\{\mathcal{I}\}$ selects interactions for execution based on the following procedure. It makes use of a non-deterministic selector (the `selector` scalar wire defined on line 10 of Listing 4.1) and selects the active interaction a_j having an index j equal to the non-deterministic value of the selector wire. If no such interaction exists, the engine selects the interaction with the largest index. This is depicted by the assignment “ $is[j] = ip[j] \wedge ((selector = j) \wedge (\neg(ip[selector] \wedge \forall k > j : \neg ip[k])))$ ” on line 32 of Listing 4.1.

$\mathcal{BIP}\{\mathcal{I}\}$ synthesizes a BIP execution engine that is based on a two clock cycles approach. In the first cycle, the engine selects an interaction $a \in \gamma$ and executes its actions. In the second cycle, the engine checks for possible transitions $\tau \in B_i$

such that $\tau.port$ is selected, for $i \in [1 \dots |I|]$, and executes their actions $\tau.actions$. Contrarily to the BIP engine in [8], $\mathcal{BIP}\{\mathcal{I}\}$'s engine executes the actions of all possible transitions simultaneously. Additionally, the engine updates the locations of the atomic components B_i of B in accordance with the executed transitions; i.e., executing a transition $\tau \in B_i.L$ transitions B_i from control location $l_1 = \tau.src$ to location $l_2 = \tau.dest$.

$\mathcal{BIP}\{\mathcal{I}\}$ updates the values of the data variables in accordance with the actions of the executed interactions and transitions. An assignment statement $\sigma \in a_k.actions$ where $a_k \in \gamma$ is an interaction executes and updates the value of its target data variable iff the interaction a_k is selected for execution, i.e., when $is[k]$ evaluate to *true* and the engine is in its first cycle. Similarly $\sigma \in \tau.actions$ where τ is a transition executes and update the value of its target data variable iff the transitions τ is allowed to execute, i.e., when $\tau.port$ is selected and the engine is in its second cycle. Updates to data variable are shown on lines 61 and 69 of the algorithm in Listing 4.1.

$\mathcal{BIP}\{\mathcal{I}\}$ uses the same AIG synthesis engine used in $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ in order to synthesize an AIG from the generated \mathcal{CAIG} program \mathcal{C} . If the developer wishes the check for deadlock freedom, $\mathcal{BIP}\{\mathcal{I}\}$ adds an assertion statement to \mathcal{C} to check that at each cycle, there is at least one interaction enabled for execution. Additionally, developers can also add custom assertions defined as FOL properties. These assertions are handled in the same manner that $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ handles them, and $\mathcal{BIP}\{\mathcal{I}\}$ uses the ABC AIG solver to check for the validity of the assertions.

4.3 Illustrative Example

Table 4.1 shows a sample execution trace of $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$'s BIP engine on the traffic light example presented in Figure 4.1. The variables $timer.t$, $timer.n$, $timer.l$, $light.l$ and $light.m$ represent t , n and the control location variable in the timer component, and the control location variable and m in the light component, respectively. We start at cycle c where that state of the BIP system is

$$(timer.t = 9, timer.n = 10, timer.l = s_0, light.l = G, light.m = 3)$$

and we assume that the execution engine has picked the interaction involving the *timer* port of the timer component has been selected and executed.

At cycle $c + 1$, the engine executes the internal transition in the timer component that corresponds to the *timer.timer* port being selected and the component is in location s_0 . Therefore the variable *timer.t* is incremented and the component remains in the same control location s_0 . Since the light component had no ports involved in the executed interaction, it will not change state.

At cycle $c + 2$, the guard $t \geq n$ is enabled and thus the *timer.done* and *light.done* ports are enabled. The engine selects the interaction connecting the two components for execution. It executes the actions associated with this interaction and transfers the value of the variable *done.m* to the variable *done.n*.

At cycle $c + 3$, the engine executes the interaction transitions that correspond to the interaction executed at cycle $c + 2$. In the timer component, the variable *timer.t*

Table 4.1: Sample of $\mathcal{BIP}\{\mathcal{I}\}$ execution

Cycle	$timer.t$	$timer.n$	$timer.l$	$light.l$	$light.m$
c	9	10	s_0	G	3
$c + 1$	10	10	s_0	G	3
$c + 2$	10	3	s_0	G	3
$c + 3$	0	3	s_0	Y	5

is reset 0 and the control location remains in location s_0 . In the light component, the variable $light.m$ takes the value 5 and the control location moves to location Y corresponding to the yellow light being displayed.

Listing 4.2 shows a code snippet from the code that $\mathcal{BIP}\{\mathcal{I}\}$ generates when applied to the traffic light controller BIP system shown in Figure 4.1. Lines 2 to 19 show the wire definitions for the system. The interaction labeled by 0 corresponds to the interaction involving the $timer.timer$ port, while the interaction labeled 1 correspond to the interaction involving the ports $timer.done$ and $light.done$. Interaction 0 is enabled ($ie[0]$) when its guard is enabled ($true$ in this case) and the port $timer.timer$ that it involves is enabled, i.e., the wire $timer.timer.e$ is set to $true$. Similarly, interaction 1 is enabled ($ie[1]$) when its guard is enabled and the ports $timer.done$ and $light.done$ are enabled ($timer.done.e \wedge light.done.e$).

The wires $ip[0]$ and $ip[1]$ are asserted when each of the two interactions is prioritized. We assume in our case that interaction 1 has a higher priority, and thus interaction 0 is only prioritized when it is enabled and interaction 1 is not. In any other case, it is interaction 1 that is prioritized by the engine.

Furthermore, the wires $is[0]$ and $is[1]$ represent the execution of the engine's interaction selection procedure. The wire $selector$ is a non-deterministic wire used by the engine to make non-deterministic choices when several interactions are prioritized. The interaction 0 is selected when it is prioritized and either the non-deterministic selector selects it ($selector == 0$) or the interaction 1 is not prioritized. The interaction 1 is selected when it is prioritized and either the selector selects it ($selector == 1$) or the selector wire has selected an interaction that is not prioritized. Since interaction 1 has the largest index, it is directly selected when the value of the selector corresponds to an interaction that is not prioritized ($\neg ip[selector]$).

The wires $component.port.e$ and $component.port.s$ represent the port enabled and selected wires, respectively. For example, the timer port in the timer component ($timer.timer.e$) is enabled when the component is in control location s_0 and the guard $timer.t < n$ is enabled. Additionally, this port is selected when the interaction 0 in which it is involved has been selected ($timer.timer.s = is[0]$).

Lines 21 to 38 show the next state functions for each of the variables and control variables of the components of the system. In the first execution cycle ($cycle == 0$), the engine selects and executes interactions. In the second execution cycle ($cycle == 1$), it executes internal transitions. When the interaction 1 is selected ($is[1]$) and the engine is in its first cycle, the value of the variable $light.m$ is transferred to the variable $timer.n$. Otherwise, $timer.n$ retains its current value.

In the internal transition execution cycle, the next state value of the variable *light.m* is dictated by the control location at which the component is currently at, and whether the port *light.done* is selected (equivalently the interaction 1 has been selected). For example, if the port *light.done* is selected and the component is at the yellow light location (*Y*), the value of *light.m* will be 10.

Finally, the control location variables *timer.l* and *light.l* are also updated according to the internal transitions of the atomic components. For example when the *light.done* port is selected and the light component is in the green control location (*G*), *light.l* moves from *G* to *Y*.

Listing 4.1: BIP to \mathcal{CAIG} transformation algorithm

```

1  Input :  $(\pi(\gamma(\{B_i\}_{i \in I})))$ , where  $\gamma = \{a_j\}_{j \in J}$ 
2  Output :  $\mathcal{CAIG}$  program  $\mathcal{C}$ 
3
4  /***** declarations *****/
5  // interaction wires
6  decl-list += wire bool ie[|J|]; // interaction enablement
7  decl-list += wire bool ip[|J|]; // interaction priority
8  decl-list += wire bool is[|J|]; // interaction selected
9
10 decl-list += wire int selector; // non-deterministic priority selector
11
12 decl-list += bool cycle; // cycle denotes whether we are executing an
    interaction or a transition
13
14 foreach i ∈ [1..|I|]
15   foreach j ∈ [1..|Bi.P|]
16     decl-list += wire bool Bi.pj.e; // port enablement
17     decl-list += wire bool Bi.pj.s; // port selected
18
19     decl-list += int Bi.ℓ;
20     foreach j ∈ [1..|Bi.X|]
21       decl-list += int Bi.xj; // variable registers
22
23 /***** wire list definitions *****/
24 foreach j ∈ [1..|J|]
25   wiredef-list += ie[j] := aj.guard ∧  $\bigwedge_{p \in a_i.P} \text{component}(p).p.e$ 
26   wiredef-list += ip[j] := ie[j] ∧  $(\forall k \neq j : ie[k] \Rightarrow a_k < a_j)$ 
27   wiredef-list += is[j] := ip[j] ∧  $(\text{selector} = j \vee (\neg ip[\text{selector}] \wedge \forall k > j : \neg ip[k]))$ 
28
29 foreach i ∈ [1..|I|]
30   foreach j ∈ [1..|Bi.P|] // where Bi.P = {p1, ..., p|Bi.P|}
31     wiredef-list += Bi.pj.e :=  $\bigvee_{\tau \in \text{transitions}(B_i.p)} \tau.\text{guard} \wedge B_i.\ell = \tau.\text{src}$ ;
32     wiredef-list += Bi.pj.s :=  $\bigvee_{a_k \in \text{interactions}(B_i.p_j)} is[k]$ ;
33
34 /***** init list definitions *****/
35 init-list += cycle := 0;
36 foreach i ∈ [1..|I|]
37   init-list += Bi.ℓ := Init.Bi;
38   foreach j ∈ [1..|Bi.X|]
39     init-list += Bi.xj := v(Bi.xj); // v is the initial valuation
40
41 /***** next list definitions *****/
42 foreach i ∈ [1..|I|]
43   foreach j ∈ [1..|Bi.X|]
44     st = Bi.xj := (cycle = 0)?
45     foreach k ∈ [1..|J|]
46       foreach assignment  $\sigma \in a_k.action$ 
47         if (Bi.xj =  $\sigma.term$ )
48           st += is[k]?  $\sigma.expr$  :
49
50     foreach  $\tau \in B_i.T$ 
51       foreach assignment  $\sigma \in \tau.action$ 
52         if (Bi.xj =  $\sigma.term$ )
53           st += Bi.port( $\tau$ ).s?  $\sigma.expr$  :
54     st += Bi.xj;
55
56     st += Bi.ℓ := (cycle = 1)?
57     foreach  $\tau \in B_i.T$ 
58       st += Bi.port( $\tau$ ).s?  $\tau.dest$  :
59     st += Bi.ℓ
60
61     next-list += st;
62 next-list += cycle :=  $\neg cycle$ ;

```

Listing 4.2: Sample of $BIP\{\mathcal{I}\}$ generated code

```

1  /** Wire definitions **/
2  ie[0] = true ∧ timer.timer.e
3  ie[1] = true ∧ timer.done.e ∧ light.done.e
4
5  ip[0] = ie[0] ∧ ¬ie[1]
6  ip[1] = ie[1]
7
8  is[0] = ip[0] ∧ ((selector == 0) ∨ (¬ip[selector] ∧ ¬ip[1]))
9  is[1] = ip[1] ∧ ((selector == 1) ∨ (¬ip[selector]))
10
11 timer.timer.e = (timer.ℓ == s0) ∧ (t < n)
12 timer.done.e = (timer.ℓ == s0) ∧ (t ≥ n)
13 light.done.e = true ∧ ((light.ℓ == G)
14                    ||(light.ℓ == R)
15                    ||(light.ℓ == Y))
16
17 timer.timer.s = is[0]
18 timer.done.s = is[1]
19 light.done.s = is[1]
20
21 /** Next state functions: Interactions **/
22 timer.n = (cycle == 0)?(is[1]?light.m : timer.n) : timer.n
23
24 /** Next state functions: Transitions **/
25 timer.t = (cycle == 1)?
26     ((timer.timer.s)?(timer.t + 1) :
27     ((timer.done.s)?0 : timer.t)) : timer.t;
28 light.m = (cycle == 1)?
29     ((light.done.s ∧ light.ℓ == G)?5 :
30     ((light.done.s ∧ light.ℓ == Y)?10 :
31     ((light.done.s ∧ light.ℓ == R)?3 : light.m)) : light.m
32 timer.ℓ = (cycle == 1)?
33     ((timer.timer.s)?s0 :
34     ((timer.done.s)?s0 : timer.ℓ)) : timer.ℓ
35 light.ℓ = (cycle == 1)?
36     ((light.done.s ∧ light.ℓ == G)?Y :
37     ((light.done.s ∧ light.ℓ == Y)?R :
38     ((light.done.s ∧ light.ℓ == R)?G : light.ℓ)) : light.ℓ

```

Chapter 5

Implementation

This chapter discusses the implementation of $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ and $\mathcal{BIP}\{\mathcal{I}\}$. It highlights the main design options and gives a brief introduction to the usage of each of the tools.

5.1 $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$

Figure 5.1 shows the overall architecture of the $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ tool. We implemented a parser for *Tiny* using ANTLR [49]. Given a program \mathcal{S} and an FOL precondition postcondition pair $(\mathcal{P}, \mathcal{Q})$, the parser generates a *control flow graph* (CFG) $G_{\mathcal{S}}$ representing the different control paths that can be taken in \mathcal{S} . The code transformation engine in $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ operates on $G_{\mathcal{S}}$ and builds a CFG $G'_{\mathcal{S}}$ in which all function calls, recursive functions, quantifiers and specifications have been resolved.

Given a bound b on the data width of variables in \mathcal{S} , $G'_{\mathcal{S}}$ is then translated into an equivalent *CAIG* CFG $G''_{\mathcal{S}}$ in accordance with the translation algorithms presented in Section 3.4. We use the AIG API provided in ABC to translate $G''_{\mathcal{S}}$ into an equisatisfiable AIG \mathcal{C} . We check the consistency and sanity of \mathcal{C} using the provided ABC checks, and can thus start performing synthesis and verification procedures. If the verification is conclusive, either a proof for $\mathcal{S} \models (\mathcal{P}, \mathcal{Q})|_b$ has been established, or a counterexample has been generated. The user can visualize the generated counterexample using the Gtkwave tool for debugging before fixing the code and re-initiating

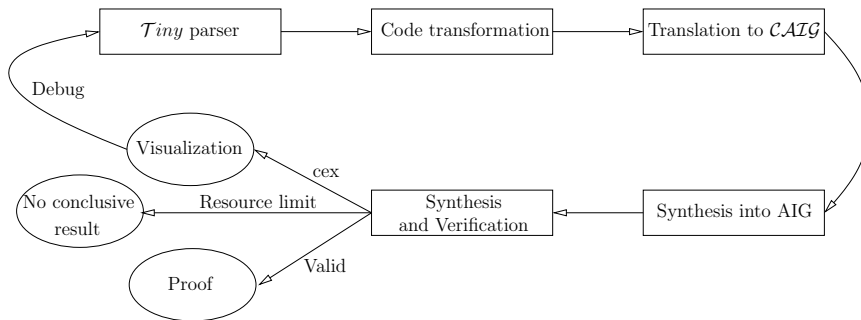


Figure 5.1: Architecture of $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$

```

1 class test {
2   int x;
3   int y;
4
5   void main ()
6   {
7       int z;
8
9       @pre add {
10        x > 0 && y > 0;
11      }
12
13      z = x + y;
14
15      @post add {
16        z > x && z > y;
17      }
18  }
19 };

```

Figure 5.2: Example source code

the verification procedure again. In the case where ABC fails to provide a conclusive result, the user can retry the verification procedures using higher computational and architectural resources if available. We implemented $\{P\}S\{Q\}$ entirely in C++.

A short tutorial

$\{P\}S\{Q\}$ is equipped with an interactive front-end that uses the GNU readline library [50] and aims at providing users with a user friendly command based interface to parse input files, start synthesis and verification procedures, and explore counter examples. Each command has a set of input arguments that can be controlled by the user to customize the synthesis and verification engines.

In addition, $\{P\}S\{Q\}$ supports a batch execution mode where it takes an input file containing a set of commands, one per line, and executes them on behalf of the user without the need for going through the interactive interface.

Figure 5.2 shows a simple example program that adds to positive integers x and y and checks that the result is larger than both. The `class` keyword is required for future support of object oriented programming, and currently has no effect on the rest of the source code. The `main` function specifies the entry of the program and is the only function that is allowed and required to be of `void` return type, and thus does not include a `return` statement.

Starting $\{P\}S\{Q\}$ in interactive mode can be done using the `--int` argument flag as follows:

```

./run --int
### This is {P}S{Q} version 1.1.0 brought to you by
### the Software Analysis and Research Lab (SARLA) at
### the American University of Beirut (AUB).

### This version was compiled on May 2 2014 08:02:00

```

```
{P}S{Q}>
```

The `help` command lists all of the available $\{P\}S\{Q\}$ commands. The description of each command can be viewed by passing it the `--help` or `-h` command line argument. We start by parsing the source code presented in figure 5.2 using the `read` command, and we pass it the name of the source file using `-i` flag:

```
{P}S{Q}> read -i add.c
{P}S{Q}::Parsing file add.c...
{P}S{Q}::Parsing successful...
```

Next we start the synthesis and verification engine using the `start_prove` command, to which we pass our customized data width bounds. $\{P\}S\{Q\}$ supports bounds on the data width of variables, on the number of elements in the arrays, and on the recursion depth of recursive functions. It also allows users to alter some flags that enable and disable array index checks, unsigned integer overflow checks and unrolling of quantifiers if possible. We start the synthesis and verification procedures with a data bit width on variables of 4 and with all checks enabled:

```
{P}S{Q}> start_prove -b 4
*** Warning! Using default recursion check depth = 3
{P}S{Q}: Removed 0 dangling nodes from code graph!
+-----+
Configuration parameters:
+-----+
Abc Debugging:           Disabled
Index Checking:          Enabled
Overflow Checking:       Enabled
Unroll quantifiers:      Yes
Group conditions:        Yes
Check termination:       No
SD Computation:          No
Latch bit width:         4
Array Bound:             15
Stack Limit:             15
Program Counter bit width: 4
+-----+
```

$\{P\}S\{Q\}$ prints the list of configuration parameters that it is going to use for synthesis and verification. The issued warning draws the user's attention to the fact that the default value for the depth of the recursion check is 3. This value is used to set a bound on the maximum depth that $\{P\}S\{Q\}$ will explore in order to determine whether a function is indirectly recursive or not.

Once the `start_prove` command terminates, $\{P\}S\{Q\}$ has generated the AIG \mathcal{C} and has linked it with ABC in order to apply reduction and proof algorithms. $\{P\}S\{Q\}$ redirects all commands that are prefixed with `abc` to the ABC model checker. Reducing and proving \mathcal{C} is done as follows:

```
{P}S{Q}> abc balance
Calling ABC with the following command
  balance
{P}S{Q}> abc ssweep -r
Calling ABC with the following command
  ssweep -r
{P}S{Q}> abc print_stats
```

```

Calling ABC with the following command
  print_stats
(null)      : i/o =   8/   1 lat =   18 and =   129 lev = 10
{P}S{Q}> abc dprove
Calling ABC with the following command
  dprove
Output 0 of miter "(null)" was asserted in frame 3. Time =   0.00 sec
Networks are not equivalent.

```

We have applied the balancing and structural register sweeping reduction algorithms on the generated AIG before calling the proof procedure `dprove`. ABC has detected a counterexample that asserts the output of \mathcal{C} in 3 time frames. We can dump the values of the counterexample in a `vcd` file that can be read by the Gtkwave waveform viewer using the `debug -oadd.vcd` command. The counterexample shows that the values of x and y are 14 and 6 respectively. Although this does not violate the postcondition \mathcal{Q} , it creates an overflow exception since that the largest representable unsigned integer with a data bit-width of 4 is 15, i.e., the addition $14 + 6$ overflows. We resort to making the precondition \mathcal{P} stronger in order to resolve the overflow. Adding the condition $x < 15 - y$ to \mathcal{P} would be enough to remove the overflow exception. We fix the source code accordingly and rerun the verification procedure using the interpolation proving algorithm:

```

{P}S{Q}> abc balance
Calling ABC with the following command
  balance
{P}S{Q}> abc ssweep -r
Calling ABC with the following command
  ssweep -r
{P}S{Q}> abc int
Calling ABC with the following command
  int
Property proved. Time =   0.05 sec

```

Running the above commands in batch mode can be easily done by saving the commands in a text file, one per line, and then passing the `--batch` command line argument along with the name of the file to $\{P\}S\{Q\}$.

Command line reference

Table 5.1 provides a reference for the command line interface options in $\{P\}S\{Q\}$. The list of command line options that can be passed to each of these commands can be retrieved by passing the `-h` flag to the command for help.

5.2 $BIP\{I\}$

We implemented $BIP\{I\}$ in JAVA and used the BIP parser provided freely in [51]. Given an input BIP program \mathcal{S} , the parser generates a parse graph G . $BIP\{I\}$ traverses G and translates \mathcal{S} into an equivalent \mathcal{CAIG} program \mathcal{S}' . We then use the same synthesis engine in $\{P\}S\{Q\}$ to synthesize an equisatisfiable AIG circuit \mathcal{C} that is passed to ABC for sequential synthesis and verification.

$BIP\{I\}$ is equipped with a command line interface that accepts a set of configuration options. It takes the name of the input BIP file, optional configuration

Table 5.1: Summary of $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ commands

Command	Description
<code>abc cmd</code>	Call ABC with the command <code>cmd</code>
<code>echo str</code>	Print <code>str</code> to the standard output
<code>help</code>	Display a list of available commands
<code>read</code>	Parse an input file and generate a code graph
<code>print_time</code>	Print the elapsed time in the last command and the total time elapsed
<code>print_conf</code>	Print the current configuration of $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$
<code>prove_status</code>	Print the status of the current proving session
<code>sim_status</code>	Print the status of the current simulation session
<code>start_prove</code>	Start a proving session
<code>start_sim</code>	Start a simulation session
<code>sim_variable</code>	Print variable values for a range of simulation frames
<code>stop_prove</code>	Stop the current proving session
<code>stop_sim</code>	Stop the current simulation session
<code>debug</code>	Debug the counter example generated from the current proving session
<code>quit,exit</code>	Exit the $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ environment

flags and an option to generate C++ emulation code. After parsing the input file, $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ performs synthesis steps and generates an output *CALG* file. We then pass this file to $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ to synthesize the AIG circuit \mathcal{C} and call ABC for reduction and verification. $\mathcal{BIP}\{\mathcal{I}\}$ makes use of the same debugging interface provided by $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ in order to allow users to visualize counterexamples if any is return by ABC.

Additionally, $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ supports the generation of C++ emulation code. It allows the users to guide the execution engine by providing a guide file that lists the order in which interactions are to take place. In case no such file is provided, the emulator uses a random number generator to choose an interaction to execute in case several candidates are available. $\mathcal{BIP}\{\mathcal{I}\}$ also allows users to set a bound on the number of clock cycles to execute in the emulation code. Furthermore, $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ allows users to specify whether to initialize the variables in the atomic components or to leave them as free inputs. Figure 5.3 shows $\mathcal{BIP}\{\mathcal{I}\}$'s command line options along with their description and an example usage.

```

##Usage
'java -jar bip-to-abc.jar [options] input.bip output.abc [property.txt]'

where:

input.bip    = input BIP file name (required)
output.abc   = ABC file name to be generated (required)
property.txt = Pre and Post condition written in two different lines (optional)

and options are:

-?           prints usage to stdout; exits (optional)
-emulator <s> Generate emulation code output.abc.cpp (optional)
              - guide.txt: indices of interactions assigned to selector
              - integer <= 0: infinite execution
              - integer > 0: number of cycles to be executed
-h           prints usage to stdout; exits (optional)
-help       displays verbose help information (optional)
-initialize-vars Initialize free variables (optional)
-version    displays command's version (optional)

##Example

'java -jar bip-to-abc.jar -initialize-vars -emulator=guide input.bip output.abc property.txt'
'java -jar bip-to-abc.jar -o -i -e=guide input.bip output.abc property.txt'
'java -jar bip-to-abc.jar -o -i -e=0 input.bip output.abc property.txt'
'java -jar bip-to-abc.jar -o -i -e=30 input.bip output.abc property.txt'
'java -jar bip-to-abc.jar -o -e=guide input.bip output.abc'

```

Figure 5.3: $BIP\{I\}$'s command line options

Chapter 6

$\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ Results

We evaluate $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ against three different program verification benchmarks. We first present a case study of linear and binary search and compare $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$'s results with those obtained from CBMC. The second set of benchmarks contains standard functions such as searching, sorting, linked list operations and array partitioning. The third set contains benchmarks from the second competition on Software Verification (SV-COMP 2013) [52], a competition aimed at the thorough evaluation of automatic program verification tools. We ran all experiments on an Intel Core i7 machine with 8 GB memory.

6.1 Searching: $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ and CBMC

The programs \mathcal{S}_1 and \mathcal{S}_2 in Figures 6.1 and 6.2 show the linear search and binary search algorithms annotated with appropriate preconditions and postconditions, respectively. The precondition in \mathcal{S}_1 ensures that the provided size n is valid, i.e., positive and smaller than the largest allowed array size. The precondition in Pm_2 checks that the provided size n is valid and that the array a is sorted in ascending order. The postcondition in both \mathcal{S}_1 and \mathcal{S}_2 checks that when the returned index rv is between 0 and the size n , then array element at the index, $a[rv]$, is equal to the element e to search for. It also checks that when rv is invalid, the element e is not present in the array a .

We use $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ and CBMC to verify that the programs \mathcal{S}_1 and \mathcal{S}_2 satisfy the precondition-postcondition pairs provided. We make use of the assume statement `__CPROVER_assume` in CBMC to implement the preconditions, and we encode the quantifiers as while loops. In $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$, we use the dedicate property grammar and the FOL support to write the specifications. We implement the check $precondition \implies postcondition$ as an assertion statement in CBMC.

We compare the results obtained from $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ and CBMC in terms of the problem size and verification time. The problem size is defined as the number of variables and clauses in the generated CNF formulae in CBMC, and the number of latches, AND gates and logic levels in the generated AIGs in $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$. We report on the size of the generated formulae and AIGs after performing optimization and reduction steps in both CBMC and $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$. We compare the time taken by both tools to perform all

```

1 int ls (int [] a, int e, int n) {
2   @pre ls {0 <= n && n <= MAX_ARRAY_SIZE}
3   int i = 0;
4   while (i < n) {
5     if (a[i] == e)
6       return i;
7     i = i + 1;
8   }
9   return -1;
10
11  @post ls { (rv >= 0 && rv < n) -> (a[rv] == e) &&
12    (rv == -1) -> (forall(int i:[0..n-1]) {a[i] != e})
13  }
14 }

```

Figure 6.1: The linear search algorithm with pre and post conditions

of the required steps to return a decision, including both optimization and verification. Since CBMC supports only 16, 32 and 64 bits scalars, we use the size of the input array a as the variable to change between different settings of the programs \mathcal{S}_1 and \mathcal{S}_2 . We set a time-out limit of 30 minutes and do not set a limit on the amount of memory the program can use (up to the machine’s physical limit).

Note that in the addition to the above, CBMC requires users to provide an *unwinding limit*. This limit is used by CBMC to unroll loops in the input programs \mathcal{S}_1 and \mathcal{S}_2 . We use an unwinding limit of $2 \times \text{array_size}$ for our experiments and we enable the CBMC unwinding assertions. $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ does not require any unwinding limit since it does not perform unrolling, it uses the program counter to execute the loops infinitely many times using the same AIG.

Table 6.1 shows the results of using $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ and CBMC to verify the programs \mathcal{S}_1 and \mathcal{S}_2 for different sizes of the input array a , shown in the column *size*. The columns *Vars* and *Clauses* report on the size of the CNF formula generated by CBMC in terms of the number of variables and clauses, respectively. The columns *lat*, *and* and *lev* show the size of the generated AIG by $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ in terms of the number of latches, and gates and logic levels, respectively. The last two columns show the time taken by $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ and CBMC to perform both reduction and verification. The *Time-out* entry indicates that the tool reached the time out limit before returning a conclusive result about the verification problem.

The results show that $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ is able to verify the linear search program \mathcal{S}_1 for sizes much higher than those provided by CBMC. For array size of 15 and above, $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ is able to efficiently generate the AIG, and call ABC to reduce and verify it, while CBMC reached the time out limit without giving any decision. Also, the size of the generated AIG by $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ is always smaller than the size of the CNF formula generated by CBMC. For example, for an array size of 15, CBMC’s CNF formula contains 9112 variables and 34496 clauses, while $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ ’s AIG has 119 bit registers and 1116 AND gates. This clearly shows the advantage that the program counter encoding provides over loop unrolling.

```

1 int bs (int [] a, int e, int n) {
2   @pre bs {0 <= n && n <= MAX_ARRAY_SIZE && isSorted(a,n)}
3   int low = 0, high = n -1;
4   while (low < high) {
5     int mid = (low + high) >> 1;
6     if (a[mid] < e) {
7       low = mid + 1;
8     } else {
9       high = mid;
10    }
11  }
12  if ((low == high) && (a[low] == e)) {
13    return low;
14  } else {
15    return -1;
16  }
17
18  @post bs { (rv >= 0 && rv < n) -> (a[rv] == e) &&
19    (rv == -1) -> (forall(int i:[0..n-1]) {a[i] != e})
20  }
21 }
22
23 isSorted(int [] a, int n) = forall(int i:[0..n-2]) {
24   a[i] <= a[i+1]
25 }

```

Figure 6.2: The binary search algorithm with pre and post conditions

For the binary search program (\mathcal{S}_2), both tools show similar performance for sizes of 3, 7 and 15. $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ outperforms CBMC for a bound of 63 since CBMC reached the time out limit while $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ was able to verify the program in 1152 seconds. Similarly to \mathcal{S}_1 , the size of the generated AIGs by $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ is orders of magnitudes smaller than the size of the generated CNF formulae by CBMC. For example, for a size of 63, the number of variables in the generated AIG is 99% smaller than the number of variables in the generated CNF formula.

6.2 Standard benchmarks

Table 6.2 shows the results of applying $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ on a set of standard program functions. The first column shows the program \mathcal{S} to verify. `ls` stands for *linear search*, `bs` stands for *binary search*, `bsort` stands for *bubble sort*, `ss` stands for *selection sort*, `a-p` stands for *array partitioning*, `lli` stands for *linked list insert* and `llr` stands for *linked list remove*. The column *b* shows the chosen bit width for the variables in \mathcal{S} . Table 6.3 lists the properties verified for each of the standard benchmarks.

To evaluate the size of the generated AIGs, we report on the number of latches (`lat`), the number of AND gates (`and`) and the number of logic levels (`lev`) as recorded by the ABC tool. We show the size of the AIGs before and after applying synthesis and

Table 6.1: $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ and CBMC comparison

		CBMC formula size		$\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ AIG size			Time (s)	
\mathcal{S}	size	Vars	Clauses	lat	and	lev	$\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$	CBMC
\mathcal{S}_1	3	2416	6784	41	313	15	4.36	0.016
\mathcal{S}_1	7	4612	15008	68	568	19	12.4	722.4
\mathcal{S}_1	15	9112	34496	119	1116	21	16.87	Time-out
\mathcal{S}_1	31	18332	84928	226	2346	24	33.67	Time-out
\mathcal{S}_1	63	37216	230208	461	5100	26	99.64	Time-out
\mathcal{S}_1	127	75876	695616	984	11315	28	396.98	Time-out
\mathcal{S}_2	3	6503	24533	56	55	19	1.04	0.085
\mathcal{S}_2	7	16172	68130	83	850	17	1.47	1.91
\mathcal{S}_2	15	42461	197223	143	1943	20	27.69	38.493
\mathcal{S}_2	63	390623	2133649	529	9052	25	1152.22	Time-out

reduction algorithms. We use common synthesis algorithms such structural sweeping (*ssweep*), retiming (*retime*), refactoring (*refactore*) and several other combinations of algorithms provided by ABC.

For performance evaluation, the column *Ver.* shows the time taken by the verification algorithm and the column *Total* shows the total time taken by ABC to perform both synthesis (reduction) and verification. The set of provided programs are all correct (i.e., contain no bugs), the *Check* column shows the result of the proof algorithm applied by ABC. A conclusive check (\checkmark) indicates that the solver was able to assert that the program satisfies its specifications. A non conclusive check (N/A) indicates that the solver hit the time limit before returning a validity answer. We set a time-out of 1800 seconds on induction based proof algorithm and a 1000000 *Binary Decision Diagrams* (BDD) size limit for algorithms based on BDD reachability. The amount of memory that $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ is allowed to use is only limited by the machine's physical limit, i.e., 8 GB.

The results in Table 6.2 clearly show the advantage that using the sequential circuit encoding provides. The ABC synthesis engine is always able to rewrite the generated AIG in a way to greatly reduce the number of AND gates and logic levels. For example, for the linear search algorithm with a bit bound of 8, ABC achieved a 93% reduction in the number of logic levels (from 529 to 33), and a 21% reduction in the number of AND gates. For more complex designs, such as the linked list insertion, the reduction algorithms achieved 50% reduction in the number of logic levels, 80% reduction in the number of AND gates and 58% reduction in the number of latches.

On average, the ABC synthesis algorithms achieved 43% reduction in the number of latches, 53% reduction in the number of AND gates and 47% reduction in the number of logic levels. Therefore we can conclude that $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ can effectively make use of ABC's synthesis algorithm to reduce the problem by half and thus help the proof algorithms. Furthermore, we note that even in the cases where the ABC solver was not able to provide a conclusive result about the properties to verify, $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ was still able to generate and efficiently reduce the AIG circuits. This in fact allows us

Table 6.2: Results of standard benchmarks

\mathcal{S}	b	Before reduction			After reduction			Time (s)		Check
		lat	and	lev	lat	and	lev	Ver.	Total	
ls	2	86	719	24	41	313	15	0.33	4.36	✓
ls	3	118	1064	27	68	568	19	3.89	12.4	✓
ls	4	174	1781	30	119	1116	21	2.41	16.87	✓
ls	5	286	3362	45	226	2346	24	1.43	33.67	✓
ls	6	526	6895	78	461	5100	26	4.57	99.64	✓
ls	7	1054	14780	143	984	11315	28	21.32	396.981	✓
ls	8	4798	70742	529	4718	55364	33	682.11	8022.11	✓
bsort	2	114	1198	29	44	393	16	0.29	5.79	✓
bsort	3	169	2218	35	68	885	20	17.1	31.09	✓
bsort	4	276	5607	47	117	2106	22	1390.25	1426.98	N/A
ss	2	112	1208	27	43	427	15	0.53	5.81	✓
ss	3	167	2239	35	69	949	19	209.37	223.54	✓
ss	4	280	5676	47	125	2236	22	1800	1852.76	N/A
a-p	2	110	1896	33	57	689	19	0.87	2.79	✓
a-p	3	147	2509	34	87	1174	24	93.47	97.56	✓
a-p	4	208	3829	38	141	2419	29	2127	2135.7	N/A
lli	2	237	4310	38	98	871	19	109.63	118.89	✓
lli	3	344	6117	41	179	1693	26	1800	1811	N/A
llr	2	197	2906	33	84	722	21	47.72	71.3829	✓
llr	3	293	4454	39	157	1387	25	1800.15	1830.21	N/A
bs	3	94	879	30	56	555	19	0.11	1.04	✓
bs	4	151	1832	42	83	850	17	0.54	1.47	✓
bs	5	268	5185	62	143	1943	20	25.42	27.69	✓

to try the validity checks using higher time-out limits and BDD size limits, or try new proof algorithms in the future.

6.3 SV-COMP 2013 benchmarks

We evaluated $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ against a select set of benchmarks obtained from the second competition on Software Verification SVCOMP'13 [52]. We selected the benchmarks from the *ControlFlowInteger* and the *Loops* and compare the execution time obtained from running $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ with the tools ranked first, second and third in each of the two categories.

Table 6.4 summarizes the results obtained from running $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ on the select set of SVCOMP'13 benchmarks. The first column shows the name of the benchmark as presented in the competition. The *status* column shows the decision that

Table 6.3: The properties checked for the standard benchmarks

\mathcal{S}	Property
Linear-search	The element is actually in the array if the return index is valid, and is not present in the array if the return index is invalid
Binary-search	The element is actually in the array if the return index is valid, and is not present in the array if the return index is invalid
Bubble-sort	The array is actually in sorted order
Selection-sort	The array is actually in sorted order
Array-partition	The array is partitioned around the element at 0
Linked-list	The list is consistent and the insertion (removal) actually took place

$\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ returned, it is colored in green to indicate that the results are accurate, i.e., $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ produced no false negative nor false positives. We report on the total execution time taken by $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$, the winner on the category (*gold*), the first runner (*silver*) and the second runner (*bronze*) for each of the selected benchmarks. Additionally, we report on the average execution time taken by each of the four tools for each of the two categories from which we selected the benchmarks. The benchmarks labeled `locks.*` belong to the *ControlFlowInteger* category, while the remaining benchmarks belong to the *Loops* category.

In the *ControlFlowInteger* category, $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ outperformed the bronze tool on all of the benchmarks. It was also able to outperform the silver tool on the first six safe benchmarks and outperform the golden tool on the first two. For the unsafe benchmarks, $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ surpassed the other tools and was able to find a counterexample 1.7 times faster than the fastest tool. On average, $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ topped the bronze tool and was ranked very closely behind the silver one.

In the *Loops* category, $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ surpassed both of the silver and the bronze tools on all of the benchmarks and came very close the gold tool. In fact, the silver tool produced a false counterexample on the `count_up_down_s` benchmark and the bronze tool produced a false counterexample on the `invert_string_s` benchmark. $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ was able to accurately verify and disprove all of the benchmarks. On average, $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ ranked second outperforming both the silver and the bronze tools, and came very closely behind the gold tool.

Table 6.4: SVCOMP'13 results

\mathcal{S}	status	time (s)	gold	silver	bronze
locks_5_safe	safe	0.28	0.4	1.2	1.3
locks_6_safe	safe	0.32	0.43	1.3	1.6
locks_7_safe	safe	0.52	0.4	1.3	1.9
locks_8_safe	safe	0.62	0.4	1.3	2.3
locks_9_safe	safe	1.05	0.39	1.3	3.5
locks_10_safe	safe	0.88	0.4	1.3	6.4
locks_11_safe	safe	3.42	0.39	1.4	24
locks_12_safe	safe	4.21	0.4	1.4	110
locks_13_safe	safe	4.18	0.43	1.4	100
locks_14_safe	safe	5.9	0.4	1.4	100
locks_15_safe	safe	6.83	0.4	1.4	100
locks_14_unsafe	unsafe	0.99	3.2	1.6	1.8
locks_15_unsafe	unsafe	0.91	4.2	1.6	1.8
Average	N/A	2.316	0.911	1.377	34.969
array_safe	safe	0.116	0.05	0.15	0.48
array_unsafe	unsafe	0.091	0.09	0.47	0.46
count_up_down_s	safe	0.137	0.28	450	0.39
count_up_down_u	unsafe	0.095	0.04	0.15	0.43
invert_string_s	safe	1.808	0.03	14	0.56
invert_string_u	unsafe	0.266	0.11	10	0.68
Average	N/A	0.419	0.100	79.128	0.500

Chapter 7

$BIP\{I\}$ Results

We evaluated $BIP\{I\}$ against two industrial benchmarks, an *Automatic Teller Machine* (ATM) [53] and the *Quorum* consensus protocol [54]. We report on the size of the generated AIGs before and after reduction, and on the time taken by the ABC solver to reduce and verify the benchmarks. We compare the results for the verification of the ATM benchmark with the NuSMV [10] model checker.

7.1 The ATM benchmark

An ATM is a computerized system that provides financial services for users in a public space. Figure 7.1 shows a structured BIP model of an ATM system adapted from the description provided in [53]. The system is composed of four atomic components: (1) the User, (2) the ATM, (3) the Bank Validation and (4) the Bank Transaction. It is the job of the ATM component to handle all interactions between the users and the bank. No communication between the users and the bank is allowed.

The ATM starts from an idle location and waits for the user to insert his card and enter the confidential code. The user has 5 time units to enter the code before the counter expires and the card is ejected by the ATM. Once the code is entered, the ATM checks with the bank validation unit for the correctness of the code. If the code is invalid, the card is ejected and no transaction occurs. If the code is valid, the

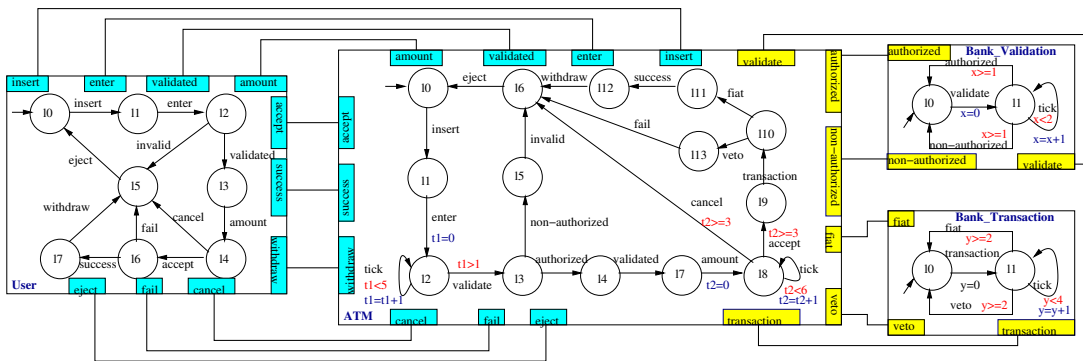


Figure 7.1: Modeling of ATM system in BIP

Table 7.1: ATM results

ATMs	Original			After reduction			Time(s)		
	lat	and	lev	lat	and	lev	Ver.	Total	NuSMV
2	78	2308	125	37	552	25	21.83	26.1	1.4
3	102	3689	197	50	804	29	32.65	38.87	142.6
4	146	5669	234	63	1036	29	590	597	3361

ATM waits for the user to enter the desired amount of money for the transaction. The time-out for entering the amount of money is of 6 time units.

Once the user enters the desired transaction amount, the ATM checks with the bank whether the transaction is allowed or not by communicating with the bank transaction unit. If the transaction is approved, the money is transferred to the user and the card is ejected. If the transaction is rejected, the user is notified and the card is ejected. In all cases, the ATM goes back to the idle location waiting for any additional users. In our model, we assume the presence of a single bank and multiple ATMs and users.

Table 7.1 shows the improvement obtained by using $\mathcal{BIP}\{\mathcal{I}\}$ to verify the deadlock freedom of the ATM system, as compared to using the NuSMV model checker [10]. The first column of the table shows the number of clients and ATMs in the system. Columns `lat`, `and` and `lev` present the number of latches, AND gates and logic levels in the AIG generated by $\mathcal{BIP}\{\mathcal{I}\}$ before and after applying reduction techniques, respectively. We report on the verification time taken by the ABC solver to check the generated AIG, and the total taken to perform both synthesis (reduction) and verification, in addition to the time taken by NuSMV to perform verification.

With the increase in the number of users and ATMs in the system, $\mathcal{BIP}\{\mathcal{I}\}$ outperforms NuSMV in terms of total verification time, reaching a speedup of 5.6 for 4 users and ATMs. Additionally, $\mathcal{BIP}\{\mathcal{I}\}$ allows developers to make use of several reduction techniques that are able to reach an average of 50% reduction in the size of the AIG. Note that for 2 ATMs and users, NuSMV outperforms $\mathcal{BIP}\{\mathcal{I}\}$. This is due to the fact that when performing verification, ABC tries multiple verification and reduction algorithms before reaching a conclusive result. However, the advantage that $\mathcal{BIP}\{\mathcal{I}\}$ presents can be clearly seen for larger number of ATMs and users.

7.2 The Quorum protocol

The *Quorum* protocol is a consensus protocol proposed in [54] as complementary to the Paxos consensus protocol [55] under perfect channel conditions. *Consensus* allows a set of communicating processes (clients and servers in our case) to agree on a common value. Each of clients proposes a value and receives a common decision value. The authors in [54] propose to use Quorum when no failures occur (perfect channel conditions) and Paxos when less than half of the servers may fail.

The Quorum protocol operates as follows.

1. Upon proposal, a client c broadcasts its proposed value v to all servers. It also

- saves v in its local memory and starts a local time t_c .
2. When a server receives a value v from a client c , it performs the following check.
 - If it has not sent any accept messages, it sends an accept message $accept(v)$ to the client c .
 - If it has already accepted value v' , it sends an accept message $accept(v')$ to the client c .
 3. If a client c receives two different accept messages, it switches to the backup phase $switch - backup(proposal_c)$.
 4. If a client c receives the same accept messages $accept(v)$ from all the servers, it decides on the value v .
 5. If a client's timer t_c expires, it waits for at least one accept message $accept(v')$ from a server, or chooses a value v' from an already received $accept(v')$ message, and then switches to the backup phase with the value v' .
 6. The *backup* phase is an implementation of the Paxos algorithm. Quorum in this case has decided that the channel is not perfect.

We implemented the Quorum protocol in BIP, and we used $\mathcal{BIP}\{\mathcal{I}\}$ to verify two invariants as defined in [54].

Invariant₁ If a client c decides on a value v , then all clients $c' \neq c$ that have switched, either before or after c , switch with the value v .

Invariant₂ If a client c decides on a value v , then all clients $c' \neq c$ who decide, do so with the same value v .

Table 7.2 shows the results of using $\mathcal{BIP}\{\mathcal{I}\}$ to verify the Quorum protocol for 2 and 4 clients with 2 servers. The designs are indexed as `num_clients-num_servers-status` where `num_clients` is the number of clients, `num_servers` is the number of servers and `status` is either valid (`v`) or erroneous (`e`). A valid design contains no design bugs, while an erroneous design is injected with a bug. We report on the size of the AIG in terms of number of latches (`lat`), number of AND gates (`and`) and logic levels (`lev`) before and after applying reduction algorithms. We also show the time taken by ABC to decide the problem, and the total time taken for reduction and decision procedures. A \checkmark decision indicates that ABC proved that the property is never violated, i.e., the design is valid, while a χ decision means that ABC was able to find a counter example that violates the property.

Using ABC's synthesis and reduction algorithms, $\mathcal{BIP}\{\mathcal{I}\}$ was able to reduce the size of the generated AIGs for all designs by a factor larger than 50%. Furthermore, $\mathcal{BIP}\{\mathcal{I}\}$ was able to give conclusive results about all four designs, unlike NuSMV which failed to give any decision about the designs having 4 clients and 2 servers. For example, $\mathcal{BIP}\{\mathcal{I}\}$ found a counter example for the erroneous design having 4 clients and 2 servers in 0.24(s) while NuSMV failed to do so. Figure 7.2 shows a snippet of the generated counter example for the erroneous design, visualized using the Gtkwave [12] waveform

Table 7.2: Quorum results

Design	Original			After reduction			Time (s)		Decision
	lat	and	lev	lat	and	lev	Ver.	Tot.	
2-2-v	264	3614	105	66	641	29	240.6	245	✓
2-2-e	264	3508	101	65	923	51	0.78	0.11	χ
4-2-v	390	6453	151	117	1170	30	58 hours		✓
4-2-e	390	6305	145	117	1129	50	0.24	0.31	χ

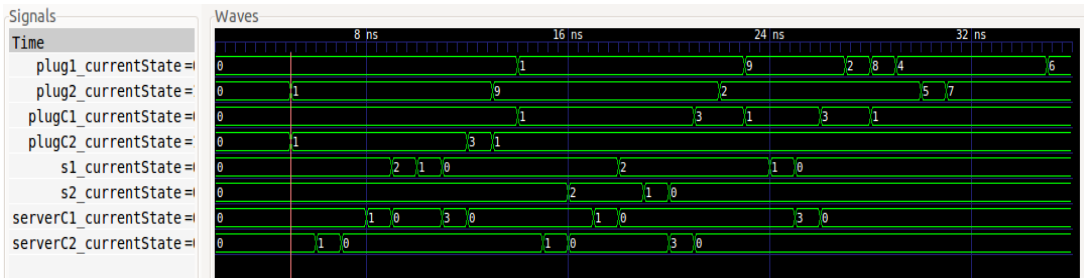


Figure 7.2: Visualization of a counter example using Gtkwave

viewer. The variables presented in the counterexample are the current control locations of the different components in the design.

Chapter 8

Related work

8.1 Verification of software programs

The need to design correct software and hardware systems has pushed researchers to design verification techniques and tools targeted towards software, hardware and embedded systems. SPIN is a model checking tool targeting the verification of process interactions. It presents a front-end with a high level specification language called Promela, aimed at allowing users to provide descriptions of concurrent systems or distributed algorithms [56]. SPIN accepts *Linear Temporal Properties* (LTL) specifications as input, and translates them into Büchi automaton using simple on-the-fly construction. In fact, SPIN considers the input specifications as impossible conditions, i.e. conditions that should never be met for correct behavior of a given system. Therefore it aims at checking whether the language of the design and that of the specification do not intersect. If such an intersection exists, SPIN returns a counter example. Otherwise the design is considered to be correct. $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ differs from SPIN in that it takes a imperative program \mathcal{S} and FOL precondition-postcondition pair $(\mathcal{P}, \mathcal{Q})$, and translates them into an equisatisfiable AIG. On the resulting AIG, several reduction and proof algorithms can be applied to perform model checking, most of which have no counterpart in SPIN.

Alloy [40] and CBMC [5] are tools that perform bounded model checking. CBMC accepts ANSI-C input code and supports several standard C libraries. It checks for pointer operations and within bound array access, and allows for dynamic memory allocation using `new` and `malloc`. CBMC takes an input program and an unwinding bound, and generates a CNF formula that describes the behavior of the program. CBMC uses *Single Static Assignment* (SSA) transformations and loop unwinding in order to generate the appropriate CNF formula.

Similarly, Alloy takes input structural specifications where entities can either be sets, functions or relations, and allows operations on these three elementary types. It then transforms such specifications into CNF formulae and uses SAT solvers to generate satisfying models. Alloy differs from CBMC in that it is aimed at generating systems that satisfy the given specifications, i.e. find models for the structural specifications, while CBMC is aimed at the verification of a given design against a set of specifications.

Both Alloy and CBMC transform the given problems into SAT problems and use an off the shelf SAT solver [57, 58, 6] to try and find a model for the generated CNF formula. The model will then be a counter example for CBMC, and a correct system for Alloy.

Techniques based on loop unwinding and translation to CNF suffer from the rapid increase in the size of the generated CNF formulae with respect to the size of the input programs and specifications, and the unwinding bounds. Additionally, the generated CNF formula may need to be regenerated in cases where the unwinding bounds were not large enough. This iterative procedure is costly and might require intractable resources.

8.2 Verification of embedded systems

The overlap between software and hardware design in embedded systems creates more challenges for the verification process. SystemC [59] is a modeling platform based on C++ that provides design abstractions at the *Register Transfer Level* (RTL), behavior, and system levels. It aims at providing a common design environment for embedded system design and hardware-software co-design. SystemC designers write their systems in C++ using SystemC class libraries that provide implementations for hardware specific objects such as concurrent modules and clocks. Therefore the input systems can be compiled using standard C++ compilers to generate binaries for simulation. SystemC allows for the communication between different components of a system through the usage of ports, interfaces and channels.

The BIP framework differs from SystemC in that it presents a dedicated language and supporting tool-set that describes the behavior of individual system components as symbolic LTS. Communication between components in BIP is ensured through ports and interactions. BIP operates at a higher level than SystemC and does not provide support for circuit level constructs.

Verification techniques for SystemC and BIP make use of symbolic model checking tools. NuSMV2 [10] is a symbolic model checker that employs both SAT and BDD based model checking techniques. It processes an input describing the logical system design as a finite state machine, and a set of specifications expressed in LTL, Computational Tree Logic (CTL) and Property Specification Language (PSL). Given a system \mathcal{S} and a set of specifications P , NuSMV2 first flattens \mathcal{S} and P by resolving all module instantiations and creating modules and processes, thus generating one synchronous design. It then performs a Boolean encoding step to eliminate all scalar variables, arithmetic and set operations and thus encode them as Boolean functions.

In order to avoid the state space explosion problem, NuSMV2 performs a cone of influence reduction [60] step in order to eliminate non-needed parts of the flattened model and specifications. The cone of influence reduction abstraction technique aims at simplifying the model in hand by only referring to variables that are of interest to the verification procedure, i.e. variables that influence the specifications to check [61].

DFinder [9] is an automated verification tool for checking invariants on systems described in the BIP language. Given a BIP system \mathcal{S} and an invariant \mathcal{I} , DFinder operates compositionally and iteratively to compute invariants \mathcal{X} of the interactions and the atomic components of \mathcal{S} . It then uses the Yices *Satisfiability Modulo Theory*

(SMT) solver [62] to check for the validity of the formula $\mathcal{X} \wedge \neg \mathcal{I} = false$. Additionally, DFinder checks the deadlock freedom of \mathcal{S} by building an invariant \mathcal{I}_d that represents the states of \mathcal{S} in which no interactions are enabled, i.e., a deadlock occurs. It then checks the for the formula $\mathcal{X} \wedge \mathcal{I}_d = false$, i.e., none of the deadlock states are reachable in \mathcal{S} .

Techniques based on symbolic model checking for the verification of BIP designs suffer from the state space explosion problem, and often fail to scale with the size and the complexity of the systems. On the other hand, DFinder does not handle data transfer between atomic components, thus limiting the range of practical applications on which it can be applied. Our technique handles data transfers and uses the wide range of synthesis and reduction algorithms provided by ABC to effectively reduce the size and the complexity of the verification problem. Most of these algorithms have no counterpart in symbolic model checking.

Chapter 9

Conclusion

In this thesis, we presented two techniques with supporting tools, $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ and $\mathcal{BIP}\{\mathcal{I}\}$, that target the verification of the imperative programs and embedded systems. $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ is a tool that takes an imperative program \mathcal{S} with a specification, a precondition and postcondition pair $(\mathcal{P}, \mathcal{Q})$, and checks whether \mathcal{S} satisfies the specification. This check is performed within a bound b on the domain of the program and specification variables ($\mathcal{S} \models (\mathcal{P}, \mathcal{Q})|_b$); i.e. when the bounded inputs of \mathcal{S} satisfy \mathcal{P} , the outputs of \mathcal{S} satisfy \mathcal{Q} . The tool translates the problem $\mathcal{S} \models (\mathcal{P}, \mathcal{Q})|_b$ into an equisatisfiable AIG. $\mathcal{BIP}\{\mathcal{I}\}$ is a tool that takes an input BIP design with a set of optional specifications, and translates it in an equisatisfiable AIG, having its own customized execution engine.

Both tools pass the generated AIGs to the AIG synthesis and verification tool ABC for reduction and model checking. ABC employs several reduction algorithms that can be used to reduce the size of the AIG and thus the size of the decision problem. ABC also includes a set of different proof algorithms that can be used to efficiently verify the AIGs generated by $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ and $\mathcal{BIP}\{\mathcal{I}\}$.

As future work, we plan to extend $\{\mathcal{P}\}\mathcal{S}\{\mathcal{Q}\}$ with support for LTL specifications, specifically *System Verilog Assertions* SVA. We will devise a synthesis algorithm that generates AIGs from SVAs and use ABC to model check designs annotated with SVAs. Furthermore, we plan to extend $\mathcal{BIP}\{\mathcal{I}\}$ with more customization to the execution engine, specifically by allowing the engine to execute multiple interactions in parallel. We envision that this parallelism will allow for the generation of more efficient AIGs, and thus improve the verification results.

Bibliography

- [1] C. Baier, J. Katoen, *et al.*, *Principles of model checking*, vol. 26202649. MIT press, 2008.
- [2] T. Kropf, *Introduction to formal hardware verification*. Springer, 1999.
- [3] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2008.
- [4] R. Brayton and A. Mishchenko, “Abc: An academic industrial-strength verification tool,” in *Computer Aided Verification*, pp. 24–40, Springer, 2010.
- [5] E. Clarke, D. Kroening, and F. Lerda, “A tool for checking ansi-c programs,” *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 168–176, 2004.
- [6] N. Sorensson and N. Een, “Minisat v1. 13-a sat solver with conflict-clause minimization,” *SAT*, vol. 2005, p. 53, 2005.
- [7] T. A. Henzinger and J. Sifakis, “The embedded systems design challenge,” in *FM 2006: Formal Methods*, pp. 1–15, Springer, 2006.
- [8] A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, and J. Sifakis, “Rigorous component-based system design using the bip framework,” *IEEE Software*, vol. 28, no. 3, pp. 41–48, 2011.
- [9] S. Bensalem, M. Bozga, T.-H. Nguyen, and J. Sifakis, “D-finder: A tool for compositional deadlock detection and verification,” in *Computer Aided Verification* (A. Bouajjani and O. Maler, eds.), vol. 5643 of *Lecture Notes in Computer Science*, pp. 614–619, Springer Berlin Heidelberg, 2009.
- [10] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, “Nusmv: a new symbolic model checker,” *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 410–425, 2000.
- [11] M. Sipser, *Introduction to the Theory of Computation*, vol. 27. Thomson Course Technology Boston, MA, 2006.
- [12] T. Bybell, “Gtkwave electronic waveform viewer,” 2010.

- [13] S. A. Edwards, “The challenges of hardware synthesis from C-like languages,” in *Design Automation and Test in Europe*, 2005.
- [14] A. Kuehlmann and J. Baumgartner, “Transformation-based verification using generalized retiming,” in *Computer Aided Verification*, pp. 104–117, Springer, 2001.
- [15] A. Kuehlmann and J. Baumgartner, “Transformation-based verification using generalized retiming,” in *Computer-Aided Verification*, July 2001.
- [16] H. Mony, J. Baumgartner, V. Paruthi, and R. Kanzelman, “Exploiting suspected redundancy without proving it,” in *Design Automation Conference*, ACM Press, 2005.
- [17] A. Kuehlmann, M. Ganai, and V. Paruthi, “Circuit-based Boolean reasoning,” in *Design Automation Conference*, pp. 232–237, June 2001.
- [18] P. Bjesse and K. Claessen, “SAT-based verification without state space traversal,” in *Formal Methods in Computer-Aided Design*, November 2000.
- [19] 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.
- [20] P. Bjesse and A. Boraly, “DAG-aware circuit compression for formal verification,” in *Int’l Conference on Computer-Aided Design*, Nov. 2004.
- [21] K. L. McMillan, *Interpolation and SAT-Based Model Checking*. Springer Berlin / Heidelberg, 2003.
- [22] D. Wang, *SAT based Abstraction Refinement for Hardware Verification*. PhD thesis, Carnegie Mellon University, May 2003.
- [23] 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.
- [24] 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.
- [25] 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.
- [26] J. Baumgartner, A. Kuehlmann, and J. Abraham, “Property checking via structural analysis,” in *Computer-Aided Verification*, July 2002.
- [27] H. Mony et al., “Scalable automated verification via expert-system guided transformations,” in *Formal Methods in Computer-Aided Design*, Nov. 04.

- [28] A. Mishchenko, M. Case, R. Brayton, and S. Jang, “Scalable and scalably-verifiable sequential synthesis,” in *Computer-Aided Design, 2008. ICCAD 2008. IEEE/ACM International Conference on*, pp. 234–241, IEEE, 2008.
- [29] P. Bjesse and A. Boralv, “Dag-aware circuit compression for formal verification,” in *Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pp. 42–49, IEEE Computer Society, 2004.
- [30] A. Mishchenko, S. Chatterjee, and R. Brayton, “Dag-aware aig rewriting a fresh look at combinational logic synthesis,” in *Proceedings of the 43rd annual Design Automation Conference*, pp. 532–535, ACM, 2006.
- [31] A. P. Hurst, A. Mishchenko, and R. K. Brayton, “Fast minimum-register retiming via binary maximum-flow,” in *Formal Methods in Computer Aided Design, 2007. FMCAD’07*, pp. 181–187, IEEE, 2007.
- [32] N. Eén and N. Sörensson, “Temporal induction by incremental sat solving,” *Electronic Notes in Theoretical Computer Science*, vol. 89, no. 4, pp. 543–560, 2003.
- [33] N. Amla, X. Du, A. Kuehlmann, R. P. Kurshan, and K. L. McMillan, “An analysis of sat-based model checking techniques in an industrial environment,” in *Correct hardware design and verification methods*, pp. 254–268, Springer, 2005.
- [34] N. Een, A. Mishchenko, and R. Brayton, “Efficient implementation of property directed reachability,” in *Formal Methods in Computer-Aided Design (FMCAD), 2011*, pp. 125–134, IEEE, 2011.
- [35] H. Mony, J. Baumgartner, V. Paruthi, and R. Kanzelman, “Exploiting suspected redundancy without proving it,” in *Proceedings of the 42nd annual Design Automation Conference*, pp. 463–466, ACM, 2005.
- [36] A. R. Bradley, “Sat-based model checking without unrolling,” in *Verification, Model Checking, and Abstract Interpretation*, pp. 70–87, Springer, 2011.
- [37] A. R. Bradley and Z. Manna, “Checking safety by inductive generalization of counterexamples to induction,” in *Formal Methods in Computer Aided Design, 2007. FMCAD’07*, pp. 173–180, IEEE, 2007.
- [38] A. Biere, *Handbook of satisfiability*, vol. 185. IOS Press, 2009.
- [39] A. Bradley and Z. Manna, *The calculus of computation: decision procedures with applications to verification*, vol. 374. Springer, 2007.
- [40] D. Jackson, “Alloy: a lightweight object modelling notation,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 11, no. 2, pp. 256–290, 2002.
- [41] 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.

- [42] 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.
- [43] Q. Zhu, N. Kitchen, A. Kuelhman, and A. Sangiovanni-Vincentelli, “Sat sweeping with local observability don’t cares,” in *Design Automation Conference*, July 2006.
- [44] F. Zaraket, A. Aziz, and S. Khurshid, “Sequential circuits for relational analysis,” in *International Conference on Software Engineering*, May 2007.
- [45] E. Torlak and D. Jackson, “Kodkod: A relational model finder,” in *Tools and Algorithms for Construction and Analysis of Systems*, March 2007.
- [46] F. A. Zaraket, A. Aziz, and S. Khurshid, “Sequential circuits for program analysis,” in *ASE*, pp. 114–123, November 2007.
- [47] E. M. Clarke, D. Kroening, and K. Yorav, “Behavioral consistency of c and verilog programs using bounded model checking,” in *Design Automation Conference, DAC*, pp. 368–371, June 2003.
- [48] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, “Scalable automated verification via expert-system guided transformations,” in *Formal Methods in Computer-Aided Design*, pp. 159–173, Springer, 2004.
- [49] T. Parr and R. Quong, “Antlr: A predicated-ll (k) parser generator,” *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995.
- [50] B. Fox and C. Ramey, “Gnu readline library, edition 2.0, for readline library version 2.0,” *Free Software Foundation, Cambridge, MA*, 1994.
- [51] Verimag, “Rigorous design of component-based systems the bip component framework.” URL: <http://www-verimag.imag.fr/Rigorous-Design-of-Component-Based.html?lang=>.
- [52] D. Beyer, “Second competition on software verification,” in *Tools and Algorithms for the Construction and Analysis of Systems* (N. Piterman and S. Smolka, eds.), vol. 7795 of *Lecture Notes in Computer Science*, pp. 594–609, Springer Berlin Heidelberg, 2013.
- [53] M. Chaudron, E. Eskenazi, A. Fioukov, and D. Hammer, “A framework for formal component-based software architecting,” in *OOPSLA*, pp. 73–80, 2001.
- [54] R. Guerraoui, V. Kuncak, and G. Losa, “Speculative linearizability,” *Acm Sigplan Notices*, vol. 47, no. 6, pp. 55–66, 2012.
- [55] E. Gafni and L. Lamport, “Disk paxos,” *Distributed Computing*, vol. 16, no. 1, pp. 1–20, 2003.
- [56] G. Holzmann, “The model checker spin,” *Software Engineering, IEEE Transactions on*, vol. 23, no. 5, pp. 279–295, 1997.

- [57] E. Goldberg and Y. Novikov, “Berkmin: A fast and robust sat-solver,” *Discrete Applied Mathematics*, vol. 155, no. 12, pp. 1549–1561, 2007.
- [58] J. Marques-Silva and K. Sakallah, “Grasp: A search algorithm for propositional satisfiability,” *Computers, IEEE Transactions on*, vol. 48, no. 5, pp. 506–521, 1999.
- [59] P. R. Panda, “Systemc: A modeling platform supporting multiple design abstractions,” in *Proceedings of the 14th International Symposium on Systems Synthesis, ISSS '01*, (New York, NY, USA), pp. 75–80, ACM, 2001.
- [60] S. Berezin, S. Campos, and E. M. Clarke, *Compositional reasoning in model checking*. Springer, 1998.
- [61] E. M. Clarke, O. Grumberg, and D. Peled, *Model checking*. MIT press, 1999.
- [62] B. Dutertre and L. De Moura, “A fast linear-arithmetic solver for dpll (t),” in *Computer Aided Verification*, pp. 81–94, Springer, 2006.