Formal Verification Using Static and Dynamic Analyses

by

Aleksandr Zaks

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Departemnt of Computer Science

New York University

May, 2007

Amir Pnueli

Acknowledgments

Foremost, I would like to thank my advisor, professor Amir Pnueli, for his patience and encouragement. His supervision of me was very liberal, yet I always felt more confident when he was around, especially during my first steps. The increased confidence was a direct result of my advisor's deep understanding of the subject and his words of encouragement. My collaboration with Amir was the highlight of my time spend at NYU, and I will cherish it the most.

Furthermore, I would like to express my gratitude to the members of the ACSys group who had to sit through many of my practice presentations. Special thanks go to Lenore Zuck with whom I worked very closely in the beginning of my studies. In addition, I want to single out Clark Barrett, who was instrumental in my career development and kindly agreed to serve as a reader on my defense committee.

I have spent two wonderful summers at NEC Labs working with System LSI group. The experience I have gained there was absolutely crucial for my understanding of the field of formal verification. I would like to thank every member of the group for their kind support and especially Franjo Ivancic, who was my mentor, colleague, and friend for the last three years. I am also thankful to Aarti Gupta for her continued support and advisement.

Besides ACSys group, there were numerous other people at NYU, who helped me to make the journey. I would like to mention Joel Spencer and Yevgeiny Dodis with whom I have interacted on several occasions from my first steps at NYU till graduation, when both served on my thesis committee.

Abstract

One of the main challenges of formal verification is the ability to handle systems of realistic size, which is especially exacerbated in the context of software verification. In this dissertation, we suggest two related approaches that, while both relying on formal method techniques, can still be applied to larger practical systems. The scalability is mainly achieved by restricting the types of properties we are considering and guarantees that are given.

Our first approach is a novel run-time monitoring framework. Unlike previous work on this topic, we expect the properties to be specified using Property Specification Language (PSL). PSL is a newly adopted IEEE P1850 standard and is an extension of Linear Temporal Logic (LTL). The new features include regular expressions and finite trace semantics, which make the new logic very attractive for run-time monitoring of both software and hardware designs. To facilitate the new logic we have extended the existing algorithm for LTL tester construction to cover the PSL-specific operators. Another novelty of our approach is the ability to use partial information about the program that is being monitored while the existing tools only use the information about the observed trace and the property under consideration. This allows going beyond the focus of traditional run-time monitoring tools – error detection in the execution trace, towards the focus of static analysis – bug detection in programs.

In our second approach, we employ static analysis to compute SAT-based function summaries to detect invalid pointer accesses. To compute function summaries, we propose new techniques for improving the precision and performance in order to reduce the false error rates. In particular, we use BDDs to represent a symbolic simulation of functions. BDDs allow an efficient representation of path-sensitive information and high level simplification. In addition, we use a light-weight range analysis technique for determining lower and upper bounds for program variables, which can further offload the work from the SAT solver. Note that while in our current implementation the analysis happens at compile time, we can also use the function summaries as a basis for run-time monitoring.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

A study by the National Institute of Standards and Technology(NIST) in 2002 found that software developers spend approximately 80 percent of their development costs on identifying and correcting software defects or bugs, resulting in an estimated 59.5 billion dollars loss to the U.S. economy. Yet, according to Carnegie Mellon University's CyLab Sustainable Computing Consortium, released commercial software typically has 20 to 30 bugs for every 1,000 lines of code. Even highly critical code such as operating system code follows a similar pattern. In Fig. 1.1, we show a bugzilla bug report for the Fedora Core operating system, with bugs grouped by severity.

The presented statistics may explain growing interest in applying formal methods towards software verification. While many techniques, like symbolic model checking, have been widely and effectively applied to hardware verification, there are few success stories when dealing with software. The main difficulty is the so-called "state

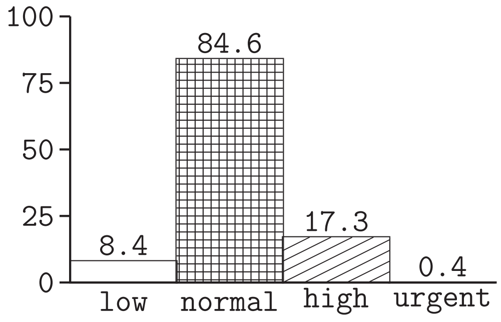

Figure 1.1: Fedora Core 5 bug report (in thousands), as of December 2006.

explosion problem". One way around the problem is to concentrate not on program correctness, the ultimate goal of formal verification, but on other important objectives. For example, one can use run-time monitoring to certify that a particular execution satisfies a given specification. Currently, there are many tools that can very efficiently validate a particular trace. Generally, there is no state explosion since the program itself is largely ignored. However, such analysis may still result in the discovery of few bugs. By using partial information about the program itself, we can hope to extend run-time monitoring to be effective for bug detection, while preserving the feasibility of the technique. The ability to effectively use partial information in run-time monitoring tools is also useful for verification of systems which use under-specified third party ("off-the-shelf") components. In such a case, directly applying model checking may lead to numerous undesirable false positives and result in a perfectly good system being declared faulty.

Many of the existing verification tools including the ones utilizing run-time monitoring use Liner Temporal Logic (LTL, see [47]) to specify the desired properties. However, recently a new, more expressive logic, called Property Specification Language (PSL, see [1]), has been accepted as IEEE P1850 standard. PSL is a language for the specification, verification, and testing of hardware design. However, it is also well suited for run-time monitoring of both hardware and software systems. First, it does include all features of LTL, including well-defined semantics for finite traces, which is very important for run-time monitoring. In addition, PSL allows using regular expressions for property specification, which are very natural for both software and hardware engineers. Thus, we would like to extend existing verification tools to work with the new specification language.

Another approach to bug detection is static analysis techniques, which have been successful in analyzing large programs for typical programming errors such as array buffer overflows, use of uninitialized variables, and others. Function summaries are often employed to enable an inter-procedural application of this technique. However, they suffer from imprecision, often resulting in high false error rates. Recent work [64] discussed how to use SAT-based techniques to compute predicated function summaries, which helped to overcome some of the resulting imprecision and reduce the false error rate. In particular, the work in [64] discussed precise analysis of locking related properties of the Linux kernel. It would be highly desirable to extend the SAT-based approach to be able to deal with more complicated bugs such as invalid pointer accesses.

1.2 Contributions

Our first contribution is a novel run-time monitoring framework. Unlike the previously known tools, we expect the properties to be specified using PSL, which is an extension of LTL. The new features include regular expressions and finite trace semantics, which makes the new logic very attractive for run-time monitoring of both software and hardware designs. To facilitate the extended language we have extended the existing algorithm for LTL tester(transducer) construction to cover the PSL specific operators [52]. As a side result, we have showed how to create an automaton that accepts the language defined by a given PSL formula. The resulting automaton is guaranteed to have fewer states than the ones produced by currently existing methods [14]. However, most importantly, since our construction is compositional, a user can submit a manually optimized tester for a sub-formula, and our tool would finish the construction for the complete formula.

Another novelty of our approach is the ability to use partial information about the program (model) that is being monitored [4] while the existing tools only use the information about the observed trace and the property under consideration. This allows going beyond the focus of traditional run-time monitoring tools – error detection in the execution trace, towards the focus of static analysis – bug detection in programs.

In our second contribution, we employ static analysis to compute SAT-based function summaries to detect invalid pointer accesses [69]. We propose new techniques for improving the precision and performance in order to reduce the false-error rates. In particular, we use BDDs to represent a symbolic simulation of functions. BDDs allow an efficient representation of path-sensitive information and high-level simplification. In addition, we use a light-weight range analysis technique for determining lower and upper bounds for program variables [2], which can further offload the work from the SAT solver. Note that while in our current implementation the analysis happens at compile time, we can also use the function summaries as a basis for run-time monitoring.

Thus, the contribution of this thesis can be summarized under the following headings:

1. PSL Testers and Optimal Monitors.
2. Run-time Monitoring with Partially Specified Systems.
3. Range Analysis.
4. Summary-Based Static Analysis for Invalid Pointer Accesses.

In the following subsections, we will elaborate on each of these results.

1.2.1 PSL Testers and Optimal Monitors

A compositional approach to the construction of automata corresponding to LTL formulas was introduced in [42], which was based on the notion of a *temporal tester* [40]. A tester for an LTL formula φ can be viewed as a *transducer* that keeps observing a state sequence σ and, at every position $j \geq 0$, outputs a boolean value which equals 1 iff $(\sigma, j) \models \varphi$ (i.e., the suffix of σ starting from position j satisfies φ) . While acceptors, such as Büchi automata [10], do not compose, transducers do. In Fig. 1.2, we show how transducers for the formulas φ, ψ, and $p\ U\ q$ can be composed into a transducer for the formula $\varphi\ U\ \psi$.

There are several important advantages to the use of temporal testers as the basis for the construction of automata for temporal formulas:

- The construction is compositional. Therefore, it is sufficient to specify testers for the basic temporal formulas: $X!p$ and $p\ U\ q$, where p and q are assertions (state formulas). Here $X!$ and U are PSL versions of the *Next* and *Until* operators, see

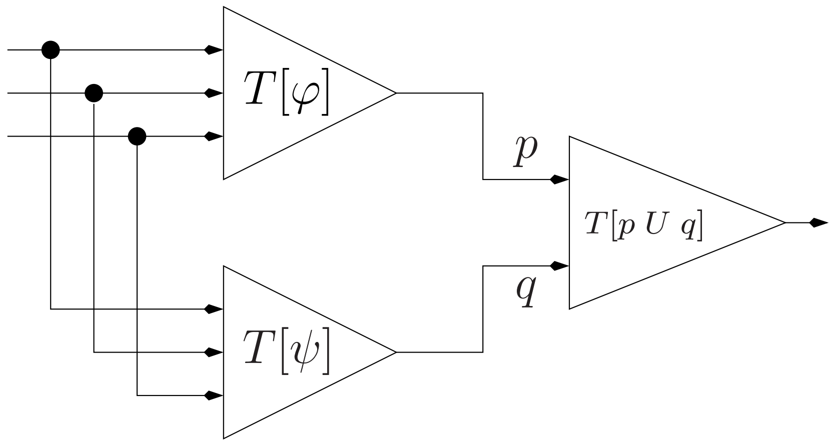

Figure 1.2: Composition of transducers to form $T[\varphi\ U\ \psi]$.

Chapter 2, Section 2.2. Testers for more complex LTL formulas can be derived by composition as in Fig. 1.2. .

- The testers for the basic formulas are naturally symbolic. Thus, a general tester, which is a synchronous parallel composition (automata product) of symbolic modules can also be easily represented symbolically.

- As will be shown in Chapter 3, Section 3.6 and Section 3.7, the basic processes of model checking and run-time monitoring can be performed directly on the symbolic representation of the testers. There is no need for determinization or reduction to explicit state representation.

In Chapter 3, we generalize the temporal tester approach to the more expressive language PSL, recently introduced as a new standard logic for specifying hardware properties [1]. Due to compositionality, it is only necessary to provide the construction of testers for the basic operators introduced by PSL.

In addition, in Chapter 3, Section 3.7 we show how to construct an optimal symbolic run-time monitor. By optimality, we mean that the monitor extracts as much information as possible from the observed trace. In particular, an optimal monitor stops as soon as it can be deduced that the specification is violated or satisfied, regardless of the possible continuations of the observed trace.

1.2.2 Run-time Monitoring with Partially Specified Systems

There are two main sources of partially specified models. First, it is commonplace, that large software systems utilize third-party components. Recently the concept of Software as a Service (SaaS) is becoming more popular, where there is even a greater separation between internal parts of a system and external components. Due to the proprietary and engineering considerations in such a situation we may not always have a complete specification (i.e., all implementation details) of all parts comprising our system. Second, we may intentionally over-approximate our system and use the resulting abstraction to increase the precision of the run-time monitor. Our hope is that we may discover more bugs while avoiding the state explosion problem associated with model checking [5] complete, fully specified, models.

As we have discussed above, out of consideration for proprietary information, or in order to simplify presentation, we should be able to deal effectively with *under-specified* models. In this dissertation, we group all under-specified components of our system and refer to it as an *interface* (environment). The rest of the system is called a *module*.

We consider the problem of a module interacting with an external interface (environment) where the interaction is expected to satisfy some system specification Φ.

While we have the full implementation details of the module, we are only given a partial external specification for the interface. The interface specification being partial (incomplete) means that the interface displays only a strict subset of the behaviors allowed by the interface specification so that directly applying model checking may result in false positives.

Based on the assumption that interface specifications are incomplete, we address the question of whether we can tighten the interface specification into a strategy, consistent with the given partial specification, that will guarantee that all possible interactions resulting from possible behaviors of the module will satisfy the system specification Φ. We refer to such a tighter specification as a Φ-*guaranteeing specification*. Rather than verifying whether the interface, which is often an off-the-shelf component, satisfies the tighter specification, the paper proposes a construction of a run-time monitor which continuously checks the existence of a Φ-guaranteeing interface. When no suitable strategy exists a monitor generates an alert indicating that there is a bug in the system.

We view the module and the external interface as players in a 2-player game. The interface has a winning strategy if it can guarantee that no matter what the module does, the overall specification Φ is met. The problem of *incomplete specifications* is resolved by allowing the interface to follow any strategy consistent with the interface specification. Our approach essentially combines traditional run-time monitoring and static analysis. This allows going beyond the focus of traditional run-time monitoring tools – error detection in the execution trace, towards the focus of the static analysis – bug detection in the programs.

In Chapter 4, we show how to construct and solve a game between the module and the interface where the solution represents the set of winning strategies for the interface. During the monitoring phase, we simply check that the monitor follows one of these strategies.

1.2.3 Range Analysis

Model checking [5] is a prominent verification technique which has been used successfully in practice to verify complex circuit designs and communication protocols. However, model checking suffers from the *state explosion* problem, i.e. the number of states to explore grows exponentially with the number of state elements. This problem is further exacerbated in the context of software verification, where variables are typically modeled as multi-bit state vectors and arrays are common.

In Chapter 5, we directly address this problem by bounding the number of bits needed to represent program variables. We statically determine possible ranges for values of variables in programs and use this information to extract smaller verification models. The use of this information greatly improves the performance of back-end model checking or static analysis techniques, based on use of BDDs or SAT solvers.

Our main method is based on the framework suggested in [55] which formulates a system of symbolic inequality constraints. It then reduces the constraint system to an LP problem, which is analyzed by an LP (Linear Programming) solver. The solution to the LP problem provides symbolic bounds for the values of all integer variables. This includes variables that are de-sugared by preprocessing steps into integer variables as described in Chapter 2, Section 2.6, in particular, pointer variables and integer (or pointer) elements of composite structures. In addition to the framework presented in [55], we have incorporated various contributions. In particular, we perform the analysis for constraint systems that can contain both disjunctions and conjunctions of linear inequalities, instead of just conjunctions. Furthermore, our particular modeling framework allows us to make certain simplifying assumptions that translate to a more efficient computation of tight bounds on variables. We use a publicly available Mixed Linear Programming (MLP) solver to solve the generated constraint systems.

Our second approach called bounded range analysis was introduced for bounded depth analyses as is done in Bounded Model Checking (BMC) [8]. Both methods have been implemented in the F-Soft verification platform [33].

1.2.4 Summary-Based Static Analysis for Invalid Pointer Accesses

Predicate abstraction [44] is a very useful technique for software verification. The general idea of predicate abstraction is that instead of tracking the values of the program variables exactly we only track the effect of program statements on a set of boolean predicates over program variables. One of the optimizations to predicate abstraction is based on the observation that there is no need to have a uniform set of predicates for all program locations. Instead, we can use a predicate localization technique [35] to reduce the overall number of tracked predicates. We can develop this idea even further and actually give up working with individual locations altogether. One possible extension is to treat a function body as a monolithic object and compute the effect of the whole function on a given set of predicates as has also been proposed in [65]. By doing so, we may significantly improve the precision of the abstraction compared to [44] since we do not have to limit our attention to a given set of predicates when processing a function. In addition, when we project the transition relation of a whole function onto the set of predicates to compute a so-called *function summary*, we expect the final result to be compact. The intuitive justification for this claim is based on a good programming practice which requires a function to represent a unit of work and have a well-defined, concise interface.

To compute function summaries, we propose many new techniques for improving the precision and performance in order to reduce the false error rates. We use BDDs to represent symbolic simulations of functions. BDDs allow an efficient representation of path-sensitive information. We propose an effective BDD variable ordering to capture the program structure, thus, avoiding a BDD size explosion as well as expensive variable reordering operations. When computing a function summary using SAT-based enumeration, we use an inter-procedural predicate clustering technique to improve the performance and scalability. We use other static analysis techniques such as program slicing to simplify and reduce a given model. To improve the overall scalability for pointer validity checking, we add context-insensitive pointer updates into caller functions to summarize the indirect effect of pointer manipulations inside a called function on aliased pointer variables. We also perform an error hierarchy analysis to limit the number of warnings presented to the user. We have implemented these techniques in the software analysis tool F-SOFT, and present experimental evidence for the efficacy of the approach.

Chapter 2

Preliminaries

2.1 Just Discrete Systems with Finite Computations

We take a *just discrete system* (JDS), which is a variant of a *fair transition system* [47], as our computational model. Under this model, a system $\mathcal{D} : \langle V, W, \Theta, R, \mathcal{J}, F \rangle$ consists of the following components:

- V: A finite set of *system variables*. A *state* of the system $\mathcal{D}$ provides a type-consistent interpretation of the system variables V. For a state s and a variable $v \in V$, we denote the value assigned to v by the state s by $s[v]$. Let S denote the set of all states over V. We assume that S is finite.

- $W \subseteq V$: A subset of *owned* variables which only the system itself can modify. All other variables can also be modified by the environment. By default, we assume our system is closed and $W = V$, in which case we omit the specification of W.

- Θ: The *initial condition*. This is an assertion (state formula) characterizing the initial states. A state is defined to be *initial* if it satisfies Θ.

- $R(V, V')$: The *transition relation*, which is an assertion that relates the values of the variables in V interpreted by a state s to the values of the variables V' in an *R-successor* state s'.

- $\mathcal{J}$: A set of *justice* (*weak fairness*) requirements. Each justice requirement is an assertion. An infinite computation must include infinitely many states satisfying the assertion.

- F: The *termination condition*, which is an assertion specifying the set of *final* states. Each finite computation must end in a final state. We sometimes omit F to indicate that we do not allow finite computations.

For a subset of variables $U \subseteq V$, we introduce the abbreviation $\mathit{pres}(U) = \bigwedge_{u \in U}(u' = u)$, specifying a transition in which all the variables in U preserve their values. If our system is open that is $W \neq V$, we define an extended transition relation $R^* = R \vee \mathit{pres}(W)$ that allows, in addition to R-steps, also environment steps. Such steps are allowed to change all variables arbitrarily, as long as they preserve the values of all owned variables.

An *open computation* of a JDS $\mathcal{D}$ is a non-empty sequence of states $\sigma : s_0, s_1, s_2, \ldots$, satisfying the requirements:

- *Initiality*: s_0 is initial.

- *Consecution*: For each $i \in [0, |\sigma|)$, where $|\sigma|$ denotes the length of σ, the state s_{i+1} is a R-successor of state s_i. That is, $\langle s_i, s_{i+1} \rangle \in R^*(V, V')$ where, for each $v \in V$, we interpret v as $s_i[v]$ and v' as $s_{i+1}[v]$.

- *Justice*: If σ is infinite, then for every $J \in \mathcal{J}$, σ contains infinitely many occurrences of J-states (i.e., states satisfying the assertion J).
- *Termination*: If $\sigma = s_0, s_1, s_2, ..., s_k$ (i.e., σ is finite), then s_k must satisfy $\mathcal{F}$. Note that sometime we only consider infinite computations, in which case $\mathcal{F} = \emptyset$, and we omit the component from a JDS description.

From now on, we will refer to an open computation simply as a "computation". A sequence of states $\sigma : s_0, s_1, s_2, ...$ that satisfies all conditions for being a computation except initiality is called an *uninitialized computation.* A sequence of states $\sigma : s_0, s_1, s_2, ...$ that only satisfies consecution is called an *uninitialized run.*

Given two JDS's $\mathcal{D}_1$ and $\mathcal{D}_2$, the systems are *compatible* if their sets of owned variables are disjoint. If the systems are compatible, their *asynchronous parallel composition*, $\mathcal{D}_1 \| \mathcal{D}_2$, is the JDS whose sets of variables, owned variables, and justice are the unions of the corresponding sets in the two systems, whose initial and termination conditions are the conjunctions of the corresponding assertions, and whose transition relation is the disjunction of the two transition relations. Thus, a step in an execution of the composed system is a step of system $\mathcal{D}_1$ or a step of system $\mathcal{D}_2$.

Similarly, for two systems $\mathcal{D}_1$ and $\mathcal{D}_2$, we define the *synchronous parallel composition*, denoted by $\mathcal{D}_1 \ ||| \ \mathcal{D}_2$, as the JDS whose sets of variables and justice requirements are the unions of the corresponding sets in the two systems, whose initial and termination conditions are the conjunctions of the corresponding assertions, and whose transition relation is defined as the conjunction of the two transition relations. Thus, a step in an execution of the composed system is a joint step of the systems $\mathcal{D}_1$ and $\mathcal{D}_2$.

Some of our examples are given in SPL (Simple Programming Language), which is used to represent concurrent programs (e.g., [47, 9]). Every SPL program can be compiled into a JDS in a straightforward manner. In particular, every statement in an SPL program contributes a disjunct to the transition relation. For example, the assignment statement "$\ell_0\colon \mathtt{x} := \mathtt{y} + \mathtt{1};\ \ell_1\colon$" contributes to the transition relation, in the JDS that describes the program, the disjunct: $at\_\ell_0 \ \wedge\ at'\_\ell_1 \ \wedge\ x' = y + 1 \ \wedge\ pres(V \setminus \{x, \pi\})$. The predicates at_ℓ_0 and at'_ℓ_1 stand, respectively, for the assertions $\pi = 0$ and $\pi' = 1$, where π is the control variable denoting the current location within the process to which the statement belongs (program counter).

2.2 Accellera PSL

In this section, we are going to follow [22] and formally define the logic PSL. However, we only consider a subset of PSL. We omit the discussions of OBE (Optional Branching Extension) formulas that are based on CTL. The branching formulas can be handled similar to [42], which describes how to combine LTL testers and CTL* branching operators. Regarding run-time monitoring, which together with model checking is the primary motivation for our work, branching formulas are not applicable at all. In addition, we only consider unclocked formulas. This is not a severe limitation since clocks do not add any expressive power to PSL [22].

2.2.1 Syntax

The logic Accellera PSL is defined with respect to a non-empty set of atomic propositions P. Let B be the set of boolean expressions over P. We assume that the expressions *true* and *false* belong to B.

Definition 1 (Sequential Extended Regular Expressions (SEREs)) .

- *Every boolean expression* $b \in B$ *is a SERE.*
- *If* $r, r_1,$ *and* r_2 *are SEREs, then the following are SEREs:*
  - $\{r\}$
  - $r_1 \,;\, r_2$
  - $r_1 : r_2$
  - $r_1 \mid r_2$
  - $[*0]$
  - $r_1 \,\&\&\, r_2$
  - $r[*]$

Definition 2 (Formulas of the Foundation Language (FL formulas)) .

- *If* r *is a SERE, then both* r *and* $r!$ *are FL formulas.*
- *If* φ *and* ψ *are FL formulas,* r *is a SERE, and* b *is a boolean expression, then the following are FL formulas:*
  - (φ)
  - $\neg\varphi$
  - $\varphi \wedge \psi$
  - $\langle r \rangle \varphi$
  - $X!\varphi$
  - $[\varphi \; U \; \psi]$
  - φ *abort* b
  - $r \mapsto \varphi$

Definition 3 (Accellera PSL Formulas) .

- *Every FL formula is an Accellera* PSL *formula.*

2.2.2 Semantics

The semantics of FL is defined with respect to finite and infinite words over $\Sigma = 2^P \cup \{\top, \bot\}$. We denote a letter from Σ by l and an empty, finite, or infinite word from Σ by u, v, or w (possibly with subscripts). We denote the length of word v as $|v|$. An empty word $v = \epsilon$ has length 0, a finite word $v = (l_0 l_1 l_2 \ldots l_k)$ has length $k+1$, and an infinite word has length ω. We use i, j, and k to denote non-negative integers. We denote the i^{th} letter of v by v^{i-1} (since counting of letters starts at zero). We denote by $v^{i..}$ the suffix of v starting at v^i. That is, for every $i <|v|$, $v^{i..} = v^i v^{i+1} \cdots v^{|v|}$ or $v^{i..} = v^i v^{i+1} \cdots$ if $|v|= \omega$. We denote by $v^{i..j}$ the finite sequence of letters starting from v^i and ending in v^j . That is, for $j \geq i$, $v^{i..j} = v^i v^{i+1} \cdots v^j$ and for $j < i, v^{i..j} = \epsilon$. We use l^ω to denote an infinite-length word, each letter of which is l.

We use $\bar{v}$ to denote the word obtained by replacing every $\top$ with a $\bot$ and vice versa. We call $\bar{v}$ the *complement* of v.

The semantics of FL *formulas* over *words* is defined inductively, using as the base case the semantics of *boolean expressions* over *letters* in Σ. The semantics of a boolean expression is assumed to be given as a relation $\Vdash \subseteq \Sigma \times B$ relating letters in Σ with boolean expressions in B. If $(l, b) \in \Vdash$, we say that the letter l satisfies the boolean expression b and denote it by $l \Vdash b$. We assume the two special letters $\top$ and $\bot$ behave as follows: for every boolean expression b, $\top \Vdash b$ and $\bot \not\Vdash b$. We assume that, otherwise, the boolean relation $\Vdash$ behaves in the usual manner. In particular, that for every letter $l \in 2^P$, atomic proposition $p \in P$ and boolean expressions $b, b_1, b_2 \in B$, (i) $l \Vdash p$ iff $p \in l$, (ii) $l \Vdash \neg b$ iff $l \not\Vdash b$, and (iii) $l \Vdash$ *true* and $l \not\Vdash$ *false*. Finally, we assume that for every letter $l \in \Sigma$, $l \Vdash b_1 \wedge b_2$ iff $l \Vdash b_1$ and $l \Vdash b_2$.

Semantics of SEREs

SEREs are interpreted over finite words from the alphabet Σ. The notation $v \models r$, where r is a SERE and v a finite word means that v *tightly models* r. The semantics of unclocked SEREs are defined as follows, where b denotes a boolean expression, and r, r_1, and r_2 denote unclocked SEREs.

- $v \models \{r\} \Longleftrightarrow v \models r$

- $v \models\!\equiv b \iff |v| = 1 \wedge v^0 \,\|\!\!= b$
- $v \models\!\equiv r_1 \,;\, r_2 \iff \exists v_1, v_2$ s.t. $v = v_1 v_2, v_1 \models\!\equiv r_1$ and $v_2 \models\!\equiv r_2$
- $v \models\!\equiv r_1 : r_2 \iff \exists v_1, v_2,$ and l s.t. $v = v_1 l v_2, v_1 l \models\!\equiv r_1$ and $l v_2 \models\!\equiv r_2$
- $v \models\!\equiv r_1 \mid r_2 \iff v \models\!\equiv r_1$ or $v \models\!\equiv r_2$
- $v \models\!\equiv r_1 \,\&\&\, r_2 \iff v \models\!\equiv r_1$ and $v \models\!\equiv r_2$
- $v \models\!\equiv [*0] \iff v = \epsilon$
- $v \models\!\equiv r[*] \iff v = \epsilon$ or $\exists v_1, v_2$ s.t. $v_1 \neq \epsilon, v = v_1 v_2$ and $v_1 \models\!\equiv r$ and $v_2 \models\!\equiv r[*]$

Semantics of FL

Let v be a finite or infinite word over Σ, b be a boolean expression, r be a SERE, and φ, ψ be FL formulas. We use $\models$ to define the semantics of FL formulas. If $v \models \varphi$ we say that v models (or satisfies) φ.

- $v \models (\varphi) \iff v \models \varphi$
- $v \models \neg\varphi \iff \bar{v} \not\models \varphi$
- $v \models \varphi \wedge \psi \iff v \models \varphi$ and $v \models \psi$
- $v \models b! \iff |v| > 0$ and $v^0 \,\|\!\!= b$
- $v \models b \iff |v| = 0$ or $v^0 \,\|\!\!= b$
- $v \models r! \iff \exists j < |v|$ s.t. $v^{0..j} \models\!\equiv r$
- $v \models r \iff \forall j < |v|,\ v^{0..j}\top^{\omega} \models r!$

- $v \models X!\varphi \iff |v| > 1$ and $v^{1..} \models \varphi$

- $v \models [\varphi\ U\ \psi] \iff \exists k < |v|$ s.t. $v^{k..} \models \psi$, and $\forall j < k,\ v^{j..} \models \varphi$

- $v \models \varphi$ *abort* $b \iff v \models \varphi$ or $\exists j < |v|$ s.t. $v^{j} \Vert\!\models b$ and $v^{0..j-1}\top^{\omega} \models \varphi$

- $v \models \langle r \rangle \varphi \iff \exists j < |v|$ s.t. $\bar{v}^{0..j} \models\!\equiv r,\ v^{j..} \models \varphi$

- $v \models r \mapsto \varphi \iff \forall j < |v|\ \bar{v}^{0..j} \models\!\equiv r,\ v^{j..} \models \varphi$

Standard LTL operators such as $\Diamond$ and $\Box$ can be easily derived from U operator.

2.3 Associating a Regular Grammar with a SERE

Following [29], a grammar $\mathcal{G} = \langle \mathcal{V}, \mathcal{T}, \mathcal{P}, \mathcal{S} \rangle$ consists of the following components:

- $\mathcal{V}$: A finite set of *variables*.

- $\mathcal{T}$: A finite set of *terminals*. We assume that $\mathcal{V}$ and $\mathcal{T}$ are disjoint. In our framework, $\mathcal{T}$ consists of boolean expressions and a special terminal ϵ.

- $\mathcal{P}$: A finite set of *productions*. We only consider right-linear grammars, so all productions are of the form $V \to aW$ or $V \to a$, where a is a terminal, and V and W are variables.

- $\mathcal{S}$: A special variable called a *start symbol*.

We say a grammar $\mathcal{G}$ is *associated* with a SERE r if, intuitively, they both define the same language. For example, we associate the following grammar $\mathcal{G}$ with SERE $r = (a_1 b_1)[*]\ \&\&\ (a_2 b_2)[*]$

$$\begin{aligned} V_1 \rightarrow \epsilon \mid \quad & (a_1 \wedge a_2)V_2 \\ V_2 \rightarrow \quad & (b_1 \wedge b_2)V_1 \end{aligned}$$

Formally, let b be a boolean expression, r', r, r_1, r_2 be SEREs, and $\mathcal{G}', \mathcal{G}, \mathcal{G}_1, \mathcal{G}_2$ the corresponding grammars. Our algorithm is recursive and we assume that $\mathcal{G}$, $\mathcal{G}_1$, and $\mathcal{G}_2$ have already been properly constructed. Our goal is to build $\mathcal{G}' = \langle \mathcal{V}', \mathcal{T}', \mathcal{P}', \mathcal{S}' \rangle$ for the SERE r'.

- $r' = b$
  - $\mathcal{V}' = \{V\}$
  - $\mathcal{T}' = \{b\}$
  - $\mathcal{P}' = \{V \rightarrow b\}$
  - $\mathcal{S}' = V$
- $r' = r_1 \,;\, r_2$
  - $\mathcal{V}' = \mathcal{V}_1 \cup \mathcal{V}_2$
  - $\mathcal{T}' = \mathcal{T}_1 \cup \mathcal{T}_2$
  - $\mathcal{P}' = \begin{array}{ll} \{V \rightarrow aW \mid V \rightarrow aW \in \mathcal{P}_1\} & \cup \\ \{V \rightarrow a\mathcal{S}_2 \mid V \rightarrow a \in \mathcal{P}_1, a \neq \epsilon\} & \cup \\ \{V \rightarrow a\mathcal{S}_2 \mid V \rightarrow aW \in \mathcal{P}_1, W \rightarrow \epsilon \in \mathcal{P}_1\} & \cup \\ \mathcal{P}_2 & \end{array}$
  - $\mathcal{S}' = \mathcal{S}_1$
- $r' = r_1 : r_2$

- $\mathcal{V}' = \mathcal{V}_1 \cup \mathcal{V}_2$
- $\mathcal{T}' = \mathcal{T}_1 \cup \mathcal{T}_2$
- $$\mathcal{P}' = \begin{array}{ll} \{V \to aW \mid V \to aW \in \mathcal{P}_1\} & \cup \\ \{V \to a \wedge b \mid V \to a \in \mathcal{P}_1, \mathcal{S}_2 \to b \in \mathcal{P}_2\} & \cup \\ \{V \to (a \wedge b)W \mid V \to a \in \mathcal{P}_1, \mathcal{S}_2 \to bW \in \mathcal{P}_2\} & \cup \\ \mathcal{P}_2 & \end{array}$$

  $$\text{where } a \wedge b = \begin{cases} \epsilon, \text{ if } a = b = \epsilon \\ a, \text{ if } b = \epsilon \\ b, \text{ if } a = \epsilon \\ a \wedge b, \text{ otherwise} \end{cases}$$
- $\mathcal{S}' = \mathcal{S}_1$

- $r' = r_1 \mid r_2$
  - $\mathcal{V}' = \{\mathcal{S}'\} \cup \mathcal{V}_1 \cup \mathcal{V}_2$
  - $\mathcal{T}' = \mathcal{T}_1 \cup \mathcal{T}_2$
  - $$\mathcal{P}' = \begin{array}{ll} \{\mathcal{S}' \to aW \mid \mathcal{S}_1 \to aW \in \mathcal{P}_1\} & \cup \\ \{\mathcal{S}' \to aW \mid \mathcal{S}_2 \to aW \in \mathcal{P}_1\} & \cup \\ \mathcal{P}_1 & \cup \\ \mathcal{P}_2 & \end{array}$$
  - $\mathcal{S}' = \mathcal{S}'$

- $r' = r_1$ && r_2
  - $\mathcal{V}' = \mathcal{V}_1 \times \mathcal{V}_2$

- $\mathcal{T}' = \mathcal{T}_1 \cup \mathcal{T}_2$

- $\mathcal{P}' = \begin{array}{ll} \{(V,X) \rightarrow a \wedge b(W,Y) \mid V \rightarrow aW \in \mathcal{P}_1, X \rightarrow bY \in \mathcal{P}_2\} & \cup \\ \{(V,X) \rightarrow a \wedge b \mid V \rightarrow a \in \mathcal{P}_1, X \rightarrow b \in \mathcal{P}_2\} & \end{array}$

- $\mathcal{S}' = (\mathcal{S}_1, \mathcal{S}_2)$

- $r' = [*0]$
  - $\mathcal{V}' = \{V\}$
  - $\mathcal{T}' = \{b\}$
  - $\mathcal{P}' = \{V \rightarrow \epsilon\}$
  - $\mathcal{S}' = V$

- $r' = r[*]$
  - $\mathcal{V}' = \mathcal{V}$
  - $\mathcal{T}' = \mathcal{T}$
  - $\mathcal{P}' = \begin{array}{ll} \{\mathcal{S} \rightarrow \epsilon\} & \cup \\ \{V \rightarrow a\mathcal{S} \mid V \rightarrow a \in \mathcal{P}, a \neq \epsilon\} & \cup \\ \{V \rightarrow a\mathcal{S} \mid V \rightarrow aW \in \mathcal{P}, W \rightarrow \epsilon \in \mathcal{P}\} & \end{array}$
  - $\mathcal{S}' = \mathcal{S}$

Theorem 1 *For every SERE r of length n, there exists an associated grammar $\mathcal{G}$ with the number of productions $O(2^n)$. If we restrict SERE's to the three traditional operators: concatenation (;), union (|), and Kleene closure ([*]), the number of productions becomes linear in the size of r.*

2.4 Game Structures

Following [18], we define a (two-player) *game* $G = (S, A, \Gamma_1, \Gamma_2, \delta)$ to consist of:

- A set S of *states*;
- A finite set A of *actions*;
- *Action assignment* functions $\Gamma_1, \Gamma_2 \colon S \to 2^A \setminus \{\emptyset\}$ that define, for each state, a non-empty set of actions available to player-1 and player-2 respectively;
- A *transition function* $\delta \colon S \times A \times A \to S$ mapping each state s and each pair of actions $(a_1, a_2) \in \Gamma_1(s) \times \Gamma_2(s)$ to a successor state $\delta(s, a_1, a_2)$;

From each state, the players simultaneously choose their actions. The two actions define the next state of the system.

Assume we are given a game G as above. For $i \in \{1, 2\}$, a *player-i strategy* is a function $\xi_i \colon S^+ \to A$ that maps every nonempty finite sequence $\bar{s} \in S^+$ to a single action that is consistent with Γ, (i.e., for every $\bar{s} \in S^*$ and $s \in S$, $\xi_i(\bar{s}; s) \in \Gamma_i(s)$). The set of strategies for player-i is denoted by Ξ_i.

Given a game structure G, a *run* r of G is a nonempty, possibly infinite, sequence $s_0(a_0^1, a_0^2)s_1(a_1^1, a_1^2)s_2 \ldots$ of alternating states and action pairs such that, for every $j \geq 0$ and $i \in \{1, 2\}$, $a_j^i \in \Gamma_i(s_j)$ and $s_{j+1} = \delta(s_j, a_j^1, a_j^2)$. For a run $r : s_0(a_0^1, a_0^2)s_1(a_1^1, a_1^2)s_2 \ldots$, we refer to the state sequence $\sigma(r) : s_0, s_1, s_2, \ldots$ as the *history induced by* r. Given a pair of strategies $\xi_1 \in \Xi_1$ and $\xi_2 \in \Xi_2$ and a state $s \in S$, the *outcome of the strategies from* s, $R_{\xi_1,\xi_2}(s)$, is a run that starts in s and whose actions are consistent with the strategies.

Let $h : s_0, s_1, \ldots, s_k{=}s$ be a finite history and Ψ a linear temporal logic formula over S. History h is said to be *a winning history* for player-i, $i \in \{1, 2\}$, with respect to objective Ψ in G if player-i has a strategy $\xi_i \in \Xi_i$ such that for all strategies $\xi_{3-i} \in \Xi_{3-i}$, $h \cdot \sigma(R_{\xi_1,\xi_2}(s)) \models \Psi$. A suitable strategy ξ_i is *a winning player-i strategy for Ψ from h in G*. In case a winning history h consists of the single state s, we refer to s as a *player-i winning state*.

Example 1 Let $S = \{s_0, s_1, s_2, s_3, s_4, s_5\}$ and $A = \{a, b, c\}$. Let Γ_1, Γ_2, and δ be defined as follows:

$$\begin{array}{llll} \Gamma_1(s_0) = \{c\}; & \Gamma_2(s_0) = \{a, b\}; & \delta(s_0, c, a) = s_1; & \delta(s_0, c, b) = s_2; \\ \Gamma_1(s_3) = \{a, b\}; & \Gamma_2(s_3) = \{c\}; & \delta(s_3, a, c) = s_4; & \delta(s_3, b, c) = s_5. \end{array}$$

For a state $s \in \{s_1, s_2, s_4, s_5\}$, $\Gamma_1(s) = \Gamma_2(s) = c$. For $j \in \{1, 2\}$, $\delta(s_j, c, c) = s_3$, and for $j \in \{4, 5\}$, $\delta(s_j, c, c) = s_j$. The corresponding game structure is shown in Fig. 2.1. Note that player-2 fully controls the transitions out of s_0. Whenever the game is at s_0, player-2 decides whether the next state will be s_1 or s_2. In a similar way, player-1 controls the exits out of s_3.

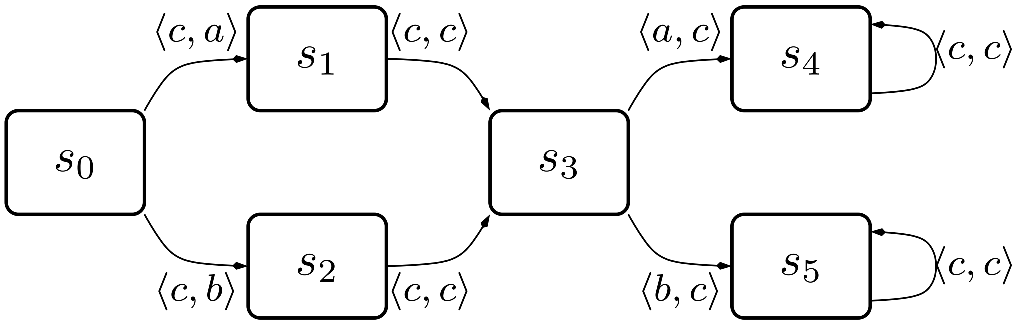

Figure 2.1: A Game structure for Example 1.

The objective of the game is defined as:

$$\Psi = \Box((s_1 \implies X!s_3 \wedge X!X!s_4) \ \wedge \ (s_2 \implies X!s_3 \wedge X!X!s_5)).$$

The objective requires that any visit to s_1 should be immediately followed by a subsequent visit to s_3, which in turn should be immediately followed by a visit to s_4, and similarly, s_2 should be followed by visits to s_3 and then to s_5.

In this game, states $\{s_3, s_4, s_5\}$ are winning for both players for Ψ. This is because no path from any of these states leads to either s_1 or s_2. The other states – s_0, s_1, and s_2 – are winning only for player-1 which, starting at any of these states, has a strategy that guarantees Ψ. Note that the winning strategy starting at s_0 depends, when we reach s_3, on the path leading to s_3, that is, on whether the previous state is s_1 or s_2. Examples of winning non-singleton histories are $h : s_0, s_1, s_3$ and $h' : s_0, s_2, s_3$, which are winning for player-1.

2.5 Rabin-Chain Automata

Assume a JDS $M = (V, W, \Theta, \rho, \mathcal{J})$. We refer to the interpretations of V (i.e., elements of S) as *computation states*. A *deterministic total Rabin-chain automaton of index k over a set of computation states S* is a tuple $R = (Q, q_0, \Delta, c)$ where:

- Q is a finite set of *automaton states*;
- $q_0 \in Q$ is the *initial state*;
- $\Delta : Q \times S \to Q$ is a transition function;
- $c\colon Q \to \{0, \dots, 2k-1\}$ is a *coloring* function.

A *run* of R over an infinite computation $\sigma : s_0, s_1, s_2, \dots,$ is an infinite sequence $q_0, q_1, q_2, \dots$ where q_0 is the initial automaton state and, for every $i \geq 0$, $\Delta(q_i, s_i) = q_{i+1}$. We say that $q_0, q_1, q_2, \dots$ is the run *induced* by the computation σ. The run

$q_0, q_1, \ldots$ is *accepting* if the maximal color that appears infinitely many times in the color sequence $c(q_0), c(q_1), \ldots$ is even. A computation σ is accepted by the automaton R if the run induced by σ is accepting. The *(ω-) language of R*, denoted by $L(R)$, is the set of computations accepted by R. We say that *the automaton R accepts the temporal formula φ* if $L(R)$ is exactly the set of computations satisfying φ.

Example 2 In Fig. 2.2, we present a Rabin-chain Automaton for the PSL formula $\Psi = \quad \Box((s_1 \implies X!s_3 \wedge X!X!s_4) \ \wedge \ (s_2 \implies X!s_3 \wedge X!X!s_5))$.

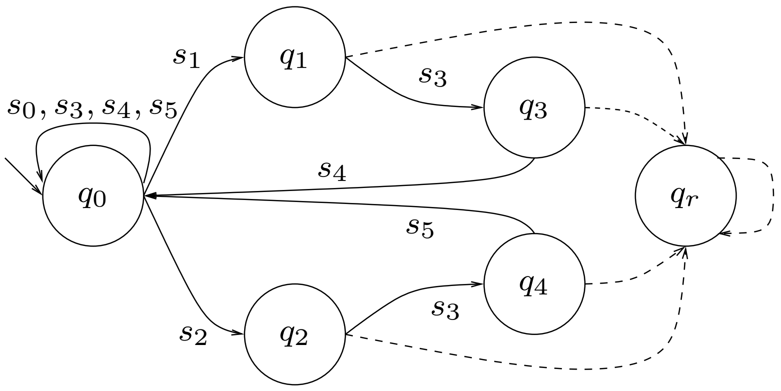

Figure 2.2: A Rabin-chain automaton for the objective Ψ of the game in Example 1

The automaton has the set of states $Q = \{q_0, q_1, q_2, q_3, q_4, q_r\}$. We connect automaton state q_i to q_j by an edge labeled by s to represent the transition function entry $\Delta(q_i, s) = q_j$. To simplify the presentation, we do not explicitly label the dashed edges which connect states to the special *rejecting state* q_r. By convention, the dashed edges are implicitly assumed to be labeled by S-states that do not label other edges departing from the same automaton state. Finally, the coloring function c is given by

$$c(q_0) = c(q_1) = c(q_2) = c(q_3) = c(q_4) = 0, \qquad c(q_r) = 1.$$

2.6 F-Soft's Software Modeling for `C` Programs

Symbolic model checkers (both SAT- and BDD-based) work on a symbolic transition relation of a finite state system, typically represented in terms of a vector of binary-valued *latches* and a Boolean next-state function (or relation) for each latch. In this section, we briefly describe an approach for translating a given `C` program, including all high-level `C` constructs such as arrays, pointers, dynamic memory, and control flow, into a Boolean model.

2.6.1 Labeled Transition Graphs

We begin with full-fledged `C` and apply a series of source-to-source transformations into smaller subsets of `C`, until program state is represented as a collection of simple scalar variables and each program step is represented as a set of parallel assignments to these variables.

Formally, the transformations produce a labeled transition graph (LTG) of the program. A LTG G is a 5-tuple $\langle B, E, X, \delta, \theta \rangle$, where

- $B = \{b_0, \ldots, b_n\}$ is a finite nonempty set of basic blocks, where b_0 is the initial block.

- $E \subseteq B \times B$ is the set of edges. If $(b_i, b_j) \in E$, we say b_i is a predecessor of b_j, and b_j is a successor of b_i.

- X is a finite set of variables that consists of actual source variables and auxiliary variables added for modeling and property monitoring.

- $\delta : B \rightarrow 2^A$ is a labeling function that labels each basic block with a set of parallel assignments taken from A. We denote a type-consistent valuation of all variables in X by $\vec{x}$, and the set of all type-consistent valuations by $\mathcal{X}$. Let the set of allowed C-expressions be denoted by Σ. Then, the parallel assignments in each basic block can be written as $Y \leftarrow e_1, \ldots, e_n$, where $Y = (y_1, \ldots, y_n), \{y_1, \ldots, y_n\} \subseteq X$ and $\{e_1, \ldots, e_n\} \subseteq \Sigma$.

- $\theta : E \rightarrow C$ is a labeling function that labels each edge with a condition from the set C. Given a basic block b_i, let $B_i \subset B$ be the set of all successors of b_i. Then $\bigvee_{b \in B_i} \theta(b_i, b) \equiv 1$.

```
int foo(int s){
  int t=s+2;
  if (t>6)
    t -= 3 ;
  else
    t--;
  return t;
}

void bar(){
  int x=3;
  int y=x-3;
  while (x<=4){
    y++ ;
    x = foo(x);
  }
  y = foo(y);
}
```

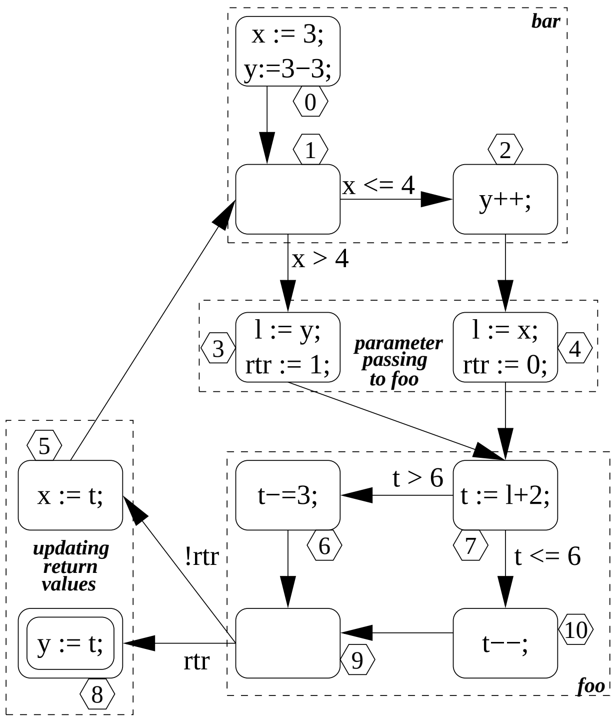

Figure 2.3: Sample code and its graph representation

As a running example, Figure 2.3 shows a LTG G obtained from the C program on the left side. Each rectangle is a basic block with an associated unique number showing its index. i.e., $B = \{b_0, \ldots, b_{10}\}$. The assignments inside each basic block are obtained by applying labeling function δ. Note that a basic block can be empty(e.g. $\delta(b_1) \equiv \emptyset$). The edges are labeled by conditional expressions; e.g. $\theta(b_1, b_2) \equiv x \leq 4$. In case a edge is not labeled by any condition, the default condition is true.The example pictorially shows how non-recursive function calls are included in the control flow of the calling function. A preprocessing analysis determines that function `foo` is not called in any recursive manner. The two return points are recorded by an encoding that passes a unique return location as a special parameter using the variable `rtr`.

2.6.2 Modeling of `C` Program Memory

One of the biggest difficulties in modeling `C` programs, lies in modeling indirect memory accesses via pointers, such as $x = *(p + i)$ or $*(q + j) = y$. This includes array accesses, since $A[e]$ is equivalent to $*(A + e)$. We replace all indirect accesses in the `C` program with expressions involving only direct variable accesses by introducing appropriate multiplexing expressions.

Modeling the heap and stack. The `C` language specification does not bound heap or stack size, but our focus is on generating a bounded model only. Therefore, we model the heap as a finite array, adding a simple implementation of `malloc()` that returns pointers into this array. We also add a bounded depth stack as another global array, in order to handle bounded recursion, if required, along with code to save and restore local state for recursive functions only.

Modeling pointers. We build an internal memory representation of the program by assigning to each variable a unique number representing its memory address. Variables that are adjacent in `C` program memory are given consecutive memory addresses in our model; this facilitates the modeling of pointer arithmetic. Pointers are modeled as integers: pointer variable p points to simple variable x by storing the integer memory address assigned to x. We perform a points-to analysis [27] to determine, for each indirect memory access, the set of variables that may be accessed (called the *points-to set*). If we determine that pointer p can point to variables $a, b, \ldots, z$ at a given program location, we can rewrite a pointer read $*(p+i)$ as a conditional multiplexing expression of the form $((p+i) == \&a ? a : ((p+i) == \&b ? b : \ldots))$ where $\&a, \&b, \ldots$ are the numeric memory addresses we assigned to the variables $a, b, \ldots$ respectively.

Inferred variables. We adopt an approach used in a hardware synthesis framework [58] by introducing additional variables when pointers are declared. For example, the declaration `int **p;` creates three variables v_p, v'_p, v''_p, where v_p stands for `p`, v'_p for `*p`, and v''_p for `**p`. In addition, a reference in the `C` code, such as `&a`, also leads to an additional variable – in this case the variable $'v_a$. In our modeling framework we may thus have several copies of the same value although they all represent the same location. Although this modeling approach may increase the number of variables when compared to the usual heap model, it can lead to analysis savings. This is due to the fact that the number of live variables can often be substantially reduced when a variable is read through a pointer dereference, because a pointer variable can point only to one location at a time, even though its points-to set may be large.

Inferred assignments. In this modeling framework additional assignments have to be inferred due to aliasing and newly introduced variables. We distinguish between two types of additional assignments, namely assignments due to aliasing and implied assignments. Assignments based on aliasing are due to the fact that multiple copies

c_1	c_2	$\cdots$	c_n	c'_1	c'_2	$\cdots$	c'_n	guard
v_1^1	v_2^1	$\cdots$	v_n^1	$v_1^{1'}$	$v_2^{1'}$	$\cdots$	$v_n^{1'}$	k^1
v_1^2	v_2^2	$\cdots$	v_n^2	$v_1^{2'}$	$v_2^{2'}$	$\cdots$	$v_n^{2'}$	k^2
	$\cdots$	$\cdots$			$\cdots$	$\cdots$		$\cdots$
v_1^m	v_2^m	$\cdots$	v_n^m	$v_1^{m'}$	$v_2^{m'}$	$\cdots$	$v_n^{m'}$	k^m

Table 2.1: Truth table for control logic

of the same location may exist. Implied assignments are due to the fact that we explicitly model the values of the pointed to locations for pointer variables.

2.6.3 Bit-accurate Model

We define a *state* of a program to be a tuple $(b, \vec{x})$, consisting of a location $b \in B$ representing the basic block, and a type-consistent valuation of data variables $\vec{x} \in \mathcal{X}$, where out-of-scope variables at b are assigned the undefined value. We consider the initial state of the program to be an initial location b_s, where each variable in X can take any value that is type-consistent with its specification.

In order to construct a Boolean model of a LTG G, we devide G into two subgraphs G_C and G_D, where G_C captures the control logic and G_D captures the data logic.

Control Logic Given a LTG $G = \langle B, E, X, \delta, \theta, \rangle$, we define a control logic subgraph $G_C = \langle B, E, X, \theta \rangle$. A program counter variable pc is introduced to monitor progress in G_C. If N denotes the number of basic blocks in a G_C, we can use $2\lceil \log N \rceil$ bits to express the program counter. Let $c_1, c_2, \cdots c_n$ denote the current state program counter bits, and $c'_1, c'_2, \cdots c'_n$ denote the next state program counter bits where $n = \lceil \log N \rceil$.

An edge $E_{ij} = (b_i, b_j)$ is enabled if and only if $pc = i \wedge \theta(b_i, b_j) = true$. Once

E_{ij} is enabled, the next value for pc is j. i.e., $pc' = j$. G_C is encoded in Table 2.1, where the first n columns show the bit values of current state pc variables, the next n columns show the bit values of next state pc variables, and the last column shows the bit value of the guarding condition. The jth table line represents an edge $(v_1^j v_2^j \cdots v_n^j \rightarrow v_1^{j'} v_2^{j'} \cdots v_n^{j'})$ in the control flow graph with guard k^j, where $v_i^j \in \{0,1\}$ is an assignment to c_i and $v_i^{j'} \in \{0,1\}$ is an assignment to c_i'. Based on the truth table, we build the next state logic for each program counter bit as:

$$c_i' = \bigvee_{j:v_i^{j'}=1} (k^j \wedge \bigwedge_{p:v_p^j=1} c_p \wedge \bigwedge_{p:v_p^j=0} \neg c_p)$$

Finally, the next state control logic for G_C is:

$$pc' \equiv \bigwedge_{i=1}^{n} c_i'$$

Data Logic A data logic subgraph of G is defined as $G_D = \langle B, X, \delta \rangle$. Assume a variable $x_i \in X$ is assigned in blocks $b_{ij} (1 \leq j \leq k)$ by expression $expr_{ij}$ and not assigned in $b_{ij} (k < j \leq n)$, the next state data logic for x_i is

$$\begin{aligned} x_i' = (&\textstyle\bigvee_{j=1}^{k} (pc = index(b_{ij})) \wedge expr_{ij}) \vee \\ (&\textstyle\bigvee_{j=k+1}^{n} (pc = index(b_{ij})) \wedge x_i), \end{aligned}$$

where $index(b)$ returns the index value of block b. The next state control logic for G_D is:

$$X' \equiv \bigwedge_{x_i \in X} x_i'$$

In order to obtain the binary logic for each variable assignment $x' = expr$, we build a combinational circuit for $expr$. For example, to handle an expression of type $expr1\&expr2$ (bitwise AND), we first build circuits for the sub-expressions $expr1$ and $expr2$. Let vectors $vec1$ and $vec2$ be the outputs of these circuits. The final result has the same bit-width as $vec1$ and $vec2$, and each result bit is the output of an AND gate with two inputs being the corresponding bits in $vec1$ and $vec2$. To handle an expression of type $expr1+expr2$, we create an n-bit adder. For the case of a relational expression the result has only one bit.

Chapter 3

PSL Testers and Optimal Monitors

3.1 Introduction

An automaton can serve as a backbone for both run-time monitoring and model checking. Indeed, the classical way of model checking an LTL property φ over a finite-state system S, represented by the automaton M_S, is based on the construction of an ω-automaton $\mathcal{A}_{\neg\varphi}$ that accepts all sequences that violate the property φ. Having both the system and its specification represented by automata, we may form the product automaton $M_S \times \mathcal{A}_{\neg\varphi}$ and check that it accepts the empty language, implying that there exists no computation of S which refutes φ [62].

Usually, the automaton $\mathcal{A}_{\neg\varphi}$ is a non-deterministic Büchi automaton, which is constructed using an explicit-state representation. In order to employ it in a symbolic (BDD-based) model checker, it is necessary to encode the automaton by the introduction of auxiliary variables. Another drawback of the normal (tableau-based) construction is that it is not compositional. That is, having constructed automata $\mathcal{A}_\varphi$ and $\mathcal{A}_\psi$ for LTL formulas φ and ψ, there is no simple recipe for constructing the automaton for a compound formula which combines φ and ψ, such as $\varphi\ U\ \psi$.

A compositional approach to the construction of automata corresponding to LTL formulas was introduced in [42], which was based on the notion of a *temporal tester* [40]. A tester for an LTL formula φ can be viewed as a *transducer* that keeps observing a state sequence σ and, at every position $j \geq 0$, outputs a boolean value which equals 1 iff $(\sigma, j) \models \varphi$. While acceptors, such as the Büchi automaton $\mathcal{A}_\varphi$, do not compose, transducers do. In Fig. 3.1, we show how transducers for the formulas φ, ψ, and $p\ U\ q$ can be composed into a transducer for the formula $\varphi\ U\ \psi$.

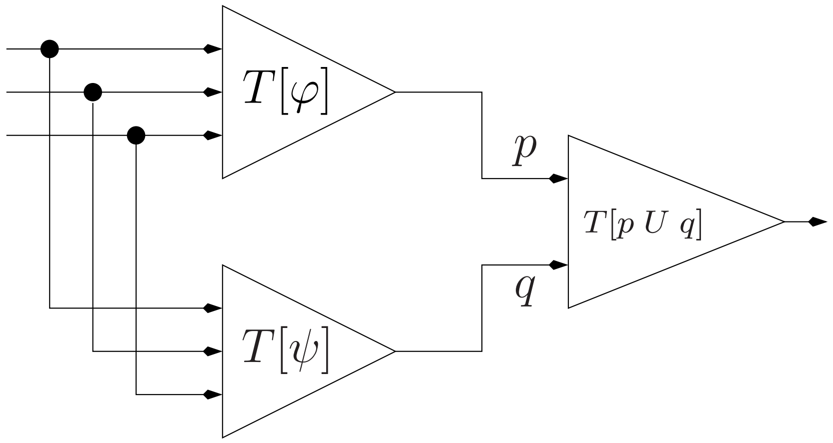

Figure 3.1: Composition of transducers to form $T[\varphi\ U\ \psi]$.

There are several important advantages to the use of temporal testers as the basis for the construction of automata for temporal formulas:

- The construction is compositional. Therefore, it is sufficient to specify testers for the basic temporal formulas: $X!p$ and $p\ U\ q$, where p and q are assertions (state formulas). Testers for more complex formulas can be derived by composition as in Fig. 3.1 .

- The testers for the basic formulas are naturally symbolic. Thus, a general tester, which is a synchronous parallel composition (automata product) of symbolic modules can also be easily represented symbolically.

- As will be shown in Chapter 3, Section 3.6 and Section 3.7, the basic processes of model checking and run-time monitoring can be performed directly on the symbolic representation of the testers. There is no need for determinization or reduction to explicit state representation.

In spite of these advantages, the complexity of constructing a transducer (temporal tester) for an arbitrary LTL formula is not worse than that of the lower-functional acceptor. In its symbolic representation, the size of a tester is linear in the size of the formula. This implies that the worst-case state complexity is exponential.

In this chapter, we generalize the temporal tester approach to the more expressive logic PSL, recently introduced as a new standard logic for specifying hardware properties [1]. Due to compositionality, it is only necessary to provide the construction of testers for the basic operators introduced by PSL.

In addition, in Section 3.7 we show how to construct an optimal symbolic run-time monitor. By optimality, we mean that the monitor extracts as much information as possible from the observed trace. In particular, an optimal monitor stops as soon as it can be deduced that the specification is violated or satisfied, regardless of the possible continuations of the observed trace.

3.2 Temporal Testers

One of the main problems in constructing a Büchi automaton for a PSL formula (or for that matter any ω-regular language) is that the conventional construction is not compositional. In particular, given Büchi automata $\mathcal{A}_\varphi$ and $\mathcal{A}_\psi$ for formulas φ and ψ, it is not trivial to build an automaton for $\varphi\ U\ \psi$. Compositionality is an important consideration, especially in the context of PSL. It is expected that specifications are written in a modular way, and PSL has several language constructs to facilitate that. For example, any property can be given a name, and a more complex property can be built by simply using a named sub-property instead of an atomic proposition.

One way to achieve compositionality with Büchi automata is to use alternation [11]. Nothing special is required from the Büchi automata to be composed in such manner, but the presence of universal branching in the resulting automaton is undesirable. Though most model checkers can deal with existential non-determinism directly and efficiently, universal branching is usually preprocessed at exponential cost.

Our approach is based on the observation that while there is very little room to maneuver during the merging step of two Büchi automata, the construction process of the sub-components is wide open for a change. In particular, we suggest that each sub-component assumes the responsibility of being easily composed with other parts. The hope is that, by requiring individual parts to be more structured than the traditional Büchi automata, we can significantly simplify the composition process.

Recall that the main property of Büchi automata (as well as any other automata) is to correctly identify a language membership of a given sequence of letters, starting from the very first letter. It turns out that for composition it is also very useful to know whether a word is in the language starting from an arbitrary position i. We refer to this new class of objects as *testers*. Essentially, testers are transducers that at each step output whether the suffix of the input sequence is in the language. Of course, the suffix is not known by the time the decision has to be made, so the testers are inherently non-deterministic.

Formally, a *tester* for a PSL formula φ is a JDS T_φ, which has a distinguished boolean variable x_φ, such that:

- **Soundness:** For every computation $\sigma : s_0, s_1, s_2, ...$ of T_φ , $s_i[x_\varphi] = 1$ iff $\sigma^{i..} \models \varphi$

- **Completeness:** For every sequence of states $\sigma' : s'_0, s'_1, s'_2, ...$, there is a matching computation $\sigma : s_0, s_1, s_2, ...$ such that for each i, s_i and s'_i agree on the interpretation of φ-variables.

Discussion of Completeness. Intuitively, the second condition requires that a tester must be able to correctly interpret x_φ for an arbitrary input sequence. Otherwise, we may essentially allow an automaton that for some input sequences outputs "I do not know". For example, a JDS that has no computations trivially satisfies the soundness condition, but cannot produce a proper output even for a single input sequence. To satisfy the completeness condition one has to ensure that the tester is not over-constrained, and it is possible to non-deterministically guess the correct values of x_φ.

3.3 LTL Testers

We are going to continue the presentation of testers by considering two very important PSL operators, namely $X!$(next) and U(until). First, we show how to build testers for two *basic formulas* $X!b$ and $b_1\ U\ b_2$, where b, b_1, and b_2 are boolean expressions. Then, we demonstrate high compositionality of the testers by easily extending the result to cover full LTL. Note that our construction for LTL operators is very similar to the one presented in [40].

3.3.1 A Tester for $\varphi = X!b$

Let $T_\varphi = \langle V_\varphi, \Theta_\varphi, R_\varphi, \mathcal{J}_\varphi, F_\varphi \rangle$ be the tester we wish to construct. The components of T_φ are defined as follows:

$$T(X!b) : \left\{ \begin{array}{rl} V_\varphi : & P \cup \{x_\varphi\},\ P \text{ is a set of propositions used to construct} B \\ \Theta_\varphi : & 1 \\ R_\varphi(V,\, V') : & x_\varphi = b' \\ \mathcal{J}_\varphi : & \emptyset \\ F_\varphi : & \neg x_\varphi \end{array} \right.$$

It almost immediately follows from the construction that $T(X!b)$ is indeed a good tester for $X!b$. The soundness of the $T(X!b)$ is guaranteed by the transition relation with the exception that we still have a freedom to incorrectly interpret x_φ at the very last state. This case is handled separately by insisting that every final state must interpret x_φ as *false*. The completeness follows from the fact that we do not restrict P variables, in any way, by the transition relation, and we can always interpret x_φ properly, by either matching b' or setting it to *false* in the last state.

3.3.2 A Tester for $\varphi = b_1 \; U \; b_2$

The components of T_φ are defined as follows:

$$T(b_1 \; U \; b_2) : \begin{cases} V_\varphi : & P \cup \{x_\varphi\} \\ \Theta_\varphi : & 1 \\ R_\varphi(V, V') : & x_\varphi = b_2 \vee (b_1 \wedge x'_\varphi) \\ \mathcal{J}_\varphi : & b_2 \vee \neg x_\varphi \\ F_\varphi : & x_\varphi = b_2 \end{cases}$$

Unlike the previous tester, $T(b_1 \; U \; b_2)$ has a non-empty justice set. A technical reason is that the transition relation allows x_φ to be continuously set to true without having a single state that actually satisfies b_2. The situation is ruled out by the justice requirement. Another way to look at the problem is that R_φ represents an expansion formula for the U(strong until) operator, namely $b_1 \; U \; b_2 \iff b_2 \vee (b_1 \wedge X![b_1 \; U \; b_2])$. In general, starting with an expansion formula is a good first step when building a tester. However, the expansion formula alone is usually not sufficient for a proper tester. Indeed, consider the operator $\mathcal{W}$(weak until), defined as $b_1 \; \mathcal{W} \; b_2 \equiv \neg(\mathit{true} \; U \; \neg b_1) \vee b_1 \; U \; b_2$, which has exactly the same expansion formula, namely $b_1 \; \mathcal{W} \; b_2 \iff b_2 \vee (b_1 \wedge X![b_1 \; \mathcal{W} \; b_2])$. We use justice to differentiate between the two operators.

3.4 Tester Composition

In Fig. 3.2, we present a recursive algorithm that builds a tester for an arbitrary LTL formula φ. In Example 3, we illustrate the algorithm by applying the tester construction for the formula $\varphi = \mathit{true} \; U \; \big(X![b_1 \; U \; b_2] \vee (b_3 \; U \; [b_1 \; U \; b_2])\big)$.

Example 3 *Tester Construction for* $\varphi = \mathit{true} \; U \; \big(X![b_1 \; U \; b_2] \vee (b_3 \; U \; [b_1 \; U \; b_2])\big)$

- **Base Case**: If φ is a basic formula (i.e., $\varphi = X!b$ or $\varphi = b_1\ U\ b_2$), use construction from Section 3.3. For a trivial case, when the formula φ does not contain any temporal operators, we can use a tester for *false* $U\ \varphi$.
- **Induction Step**: Let ψ be an innermost basic sub-formula of φ, then $T_\varphi = T_{\varphi[\psi/x_\psi]}\ \ |||\ \ T_\psi$, where $\varphi[\psi/x_\psi]$ denotes the formula φ in which each occurrence of the sub-formula ψ is replaced with x_ψ.

Figure 3.2: Tester construction for an arbitrary LTL formula φ

We start by identifying $b_1\ U\ b_2$ to be the innermost basic sub-formula and building the corresponding tester, $T_{b_1Ub_2}$. Assume that z is the output variable of the tester $T_{b_1Ub_2}$. Let $\alpha = \varphi[b_1\ U\ b_2/z]$; after the substitution $\alpha = \mathit{true}\ U\ \big(X!z \vee (b_3\ U\ z)\big)$. Note that we performed the substitution twice, but there is no need for two testers, which can result in significant savings. We proceed in similar fashion and build two more testers $T_{X!z}$ and T_{b_3Uz} with the output variables x and y. After the substitutions, we obtain $\beta = \mathit{true}\ U\ [x \vee y]$. Since $x \vee y$ is just a boolean expression, the formula satisfies the condition of the base case, and we can finish the construction with one more step. The final result can be expressed as:

$$T_\varphi = T_\beta\ |||\ T_{X!z}\ |||\ T_{b_3Uz}\ |||\ T_{b_1Ub_2}.$$

Though we have assumed φ is an LTL formula, the algorithm is applicable for PSL as well. The only extension necessary is the ability to deal with additional basic formulas.

3.5 PSL Testers

As we have mentioned before, to handle the full PSL it is enough to handle all the basic PSL formulas. More complicated formulas can be handled via tester composition according to the algorithm in Fig. 3.2. There are only two additional basic PSL formulas that we need to consider, namely $\varphi = \langle r \rangle b$ and $\varphi = r$, where r is a SERE and b is a boolean expression. All other PSL temporal operators can be expressed using those two and the LTL operators, $X!$ and U. For example, $r! \equiv \langle r \rangle \mathit{true}$, and $r \mapsto \varphi \equiv \neg(\langle r \rangle \neg\varphi)$. The *abort* operator is a little bit more complicated, and we present a set of rewriting rules in Section 3.5.3.

3.5.1 A Tester for $\varphi = \langle r \rangle b$

Let $T_\varphi = \langle V_\varphi, \Theta_\varphi, R_\varphi, \mathcal{J}_\varphi, F_\varphi \rangle$ be the tester we wish to construct. Assume that x_φ is the output variable. Let $\mathcal{G} = \langle \mathcal{V}, \mathcal{T}, \mathcal{P}, \mathcal{S} \rangle$ be a grammar associated with r. Without the loss of generality, we assume $\mathcal{G}$ has variables $V_1, \ldots, V_n$ with V_1 being the start symbol. In addition, each variable V_i, has derivations of the form:

$$V_i \rightarrow \alpha_1 \mid \cdots \mid \alpha_m \mid \beta_1 V_1 \mid \cdots \mid \beta_n V_n$$

where $\alpha_1, \ldots, \alpha_m, \beta_1, \ldots, \beta_n$ are boolean expressions. The case that variable V_i does not have a particular derivation $V_i \rightarrow \beta_j V_j$ or $V_i \rightarrow \alpha_k$, is covered by having $\beta_j = \mathit{false}$, and similarly, $\alpha_k = \mathit{false}$. Note that by insisting on this specific form, which does not allow ϵ productions, we can not express whether an empty string is in the language. However, since, by definition of $\langle\rangle$ operator, a prefix that satisfies r must be non-empty, we do not need to consider this. The tester T_φ is given by:

$$T(\langle r\rangle b) : \left\{ \begin{array}{rl} V_\varphi : & P \cup \{x_\varphi\} \cup \{X_1, \ldots, X_n, Y_1, \ldots, Y_n\} \\ \Theta_\varphi : & 1 \\ R_\varphi(V, V') : & \text{Each derivation } V_i \rightarrow \alpha_1 \mid \cdots \mid \alpha_m \mid \beta_1 V_1 \mid \cdots \mid \beta_n V_n \\ & \text{contributes to } R_\varphi \text{ the conjunct} \\ & X_i = (\alpha_1 \wedge b) \vee \cdots \vee (\alpha_m \wedge b) \vee (\beta_1 \wedge X_1') \vee \cdots \vee (\beta_n \wedge X_n') \\ & \text{and the conjunct} \\ & Y_i \rightarrow (\alpha_1 \wedge b) \vee \cdots \vee (\alpha_m \wedge b) \vee (\beta_1 \wedge Y_1') \vee \cdots \vee (\beta_n \wedge Y_n') \\ & \text{the output variable is constrained by the conjunct} \\ & x_\varphi = X_1 \\ \mathcal{J}_\varphi : & \{\neg Y_1 \wedge \cdots \wedge \neg Y_n, \quad X_1 = Y_1 \wedge \cdots \wedge X_n = Y_n\} \\ F_\varphi : & \text{Each derivation } V_i \rightarrow \alpha_1 \mid \cdots \mid \alpha_m \mid \beta_1 V_1 \mid \cdots \mid \beta_n V_n \\ & \text{contributes to } F \text{ the conjunct} \\ & X_i = (\alpha_1 \wedge b) \vee \cdots \vee (\alpha_m \wedge b) \end{array} \right.$$

Figure 3.3: A tester for $\varphi = \langle r\rangle b$.

Example 4 *A Tester for* $\varphi = \langle\{pq\}[*]\rangle b$.

To illustrate the construction, consider formula $\langle\{pq\}[*]\rangle b$. Following the algorithm for grammar construction given in Chapter 2, Section 2.3 and removing ϵ productions, the associated right-linear grammar for the SERE $\{pq\}[*]$ is given by

$$\begin{array}{ll} V_1 \rightarrow & pV_2 \\ V_2 \rightarrow q \mid & qV_1 \end{array}$$

Consequently, a tester for $\langle\{pq\}[*]\rangle b$ is given by

$$
T(\langle\{pq\}[*]\rangle b): \left\{
\begin{array}{rl}
V_\varphi : & P \cup \{x_\varphi\} \cup \{X_1, X_2, Y_1, Y_2\} \\
\Theta_\varphi : & 1 \\
R_\varphi(V, V') : & \left(\begin{array}{ll}
(X_1 = (p \wedge X_2')) & \wedge \\
(X_2 = (q \wedge b) \vee (q \wedge X_1')) & \wedge \\
(Y_1 \rightarrow (p \wedge Y_2')) & \wedge \\
(Y_2 \rightarrow (q \wedge b) \vee (q \wedge Y_1')) & \wedge \\
x_\varphi = X_1 &
\end{array}\right) \\
\mathcal{J}_\varphi : & \{\neg Y_1 \wedge \neg Y_2, \quad X_1 = Y_1 \wedge X_2 = Y_2\} \\
F_\varphi : & (X_1 = \mathit{false}) \wedge (X_2 = q \wedge b)
\end{array}\right.
$$

The variables $\{X_1, \ldots, X_n, Y_1, \ldots, Y_n\}$ are expected to check that the rest of the sequence from now on has a prefix satisfying the SERE r. Thus, the subsequence $s_j, \ldots, s_k, \ldots \models \langle r\rangle b$ iff there exists a generation sequence $V^j = V_1, V^{j+1}, \ldots, V^k$, such that for each $i, j \leq i < k$, there exists a grammar rule $V^i \rightarrow \beta V^{i+1}$, where $s_i \Vdash \beta, V^k \rightarrow \alpha,$ and $s_k \Vdash (\alpha \wedge b)$.

The generation sequence is represented in a run of the tester by a sequence of true valuations for the variables $Z^j = Z_1, Z^{j+1}, \ldots, Z^k$ where $Z^i \in \{X^i, Y^i\}$ for each $i \in [j..k]$. An important element in this checking is to make sure that any such generation sequence is finite. This is accomplished through the double representation of each V_i by X_i and Y_i. The justice requirement $(X_1 = Y_1) \wedge \cdots \wedge (X_n = Y_n)$ guarantees that any true X_i is eventually copied into Y_i. The justice requirement $\neg Y_1 \wedge \cdots \wedge \neg Y_n$ guarantees that all true Y_i's are eventually falsified. Together, they guarantee that there exists no infinite generation sequence. The double representation approach was first introduced in [50].

Theorem 2 *The* JDS $T(\langle r\rangle b)$ *in Fig. 3.3 is a proper tester for the formula* $\varphi = \langle r\rangle b$.

A proof is presented below.

First, lets remind ourselves the semantic definition of the formula φ.

$$\sigma \vDash \langle r \rangle b \Longleftrightarrow \exists j < |\sigma| \text{ s.t. } \bar{\sigma}^{0..j} \models\!\equiv r,\ \sigma^{j} \Vert\!\models b.$$

Since computations of a tester do not contain $\top, \bot$ states the bar can be removed, so that the definition is simplified into

$$\sigma \vDash \langle r \rangle b \Longleftrightarrow \exists j < |\sigma| \text{ s.t. } \sigma^{0..j} \models\!\equiv r,\ \sigma^{j} \Vert\!\models b.$$

Soundness:

If $\sigma : s_0, s_1, s_2, ...,$ is a computation of T_φ then for every $i < |\sigma|$, $s_i[x_\varphi] = 1$ iff $\sigma^{i..} \models \varphi$.
$\underline{\sigma^{i..} \models \varphi \Rightarrow s_i[x_\varphi] \text{ direction.}}$

Let s_i be a state such that $\sigma^{i..} \models \varphi$, but x_φ is incorrectly interpreted as false. Since $\sigma^{i..} \models \varphi$, let s_j be a state such that $\sigma^{i..j} \models\!\equiv r \wedge s^j \Vert\!\models b$. In addition, there must be a derivation sequence $V^i = V_1, V^{i+1}, \ldots, V^j$ such that:

- for each $k \in [i, j)$, there is a derivation rule $V^k \rightarrow \beta^k V^{k+1}$ such that $s_k \Vert\!\models \beta^k$.
- there is a derivation rule $V^j \rightarrow \alpha$ such that $s_i \Vert\!\models \alpha$

Let $X^i = X_1 = x_\varphi, X^{i+1}, \ldots, X^j$ be the evaluation of X variables that correspond to $V^i = V_1, V^{i+1}, \ldots, V^j$. Since $s_i[x_\varphi]$ is assumed to be false (i.e., X^i is false), according to the transition relation and the existence of the derivation sequence, it must be the case that the variable X^{i+1} that corresponds to V^{i+1} must also be set to false. We can continue applying the transition relation to conclude that the variable X^j is false.

However, this contradicts to the transition relation (or termination condition if s_j is the very last state) since $s_i \not\models \alpha \wedge b$.

$s_i[x_\varphi] \Rightarrow \sigma^{i..} \models \varphi$ direction.

Let s_i be a state such that $\sigma^{i..} \not\models \varphi$, but we is incorrectly interpreted as true. Note that s_i cannot be the last state of the computation; otherwise, we immediately reach a contradiction due to the termination condition. To proceed from s_i to s_{i+1} we must obey transition relation. In particular, there must an X-variable, set to true. We will refer to this variable as X^{i+1} since we are taking about state s_{i+1}. Following an *alpha*-based disjunct is not option due to the assumption that $\sigma^{i..} \not\models \varphi$, and applying an *alpha*-based disjunction would imply a successful derivation in one step. Continuously applying the above reasoning we get an infinite sequence $X^i = X_1 = x_\varphi, X^{i+1}, X^{i+2}, \ldots$ where the mentioned Y variables are interpreted as true by the corresponding states According to the second justice requirement that must be a position j such that $X^j = Y^j$ where both Y^j and X^j represent the same non-terminal V.

We can now apply the same reasoning for Y^j as we did for X^i, since for true valuations of Y variables the transition relation is the same as for X. Therefore, we have obtained an infinite sequence $Y^j = X^j, Y^{i+1}, Y^{i+2}, \ldots$, where the mentioned Y variables are interpreted as true. This trivially contradicts the first justice requirement.

Completeness:

For every sequence of states $\widetilde{\sigma} : \widetilde{s}_0, \widetilde{s}_1, \widetilde{s}_2, ...,$, there is a matching computation $\sigma : s_0, s_1, s_2, ...$ such that for each k, s_k and $\widetilde{s}_k$ agree on the interpretation of φ-variables.

Below is a constructive proof that shows how to build a σ from $\widetilde{\sigma}$.

- A state s_k interprets X_i as true iff $\sigma^{k..} \models \langle V_i \rangle b$. By $\langle V_i \rangle$ we mean a regular expression with a V_i being a starting symbol. Using this new notation, we can express our original formula $\varphi = \langle r \rangle b$ as $\varphi = \langle V_1 \rangle b$;

- All Y variables are interpreted as false by the initial state s_0;

- If all Y variables are interpreted as false by a state s_k, we copy values of X variables into the corresponding Y variables (i.e., s_{k+1} interprets Y_i the same way as X_i);

- If a state s_k interprets Y_i as true, and there is no applicable α-based rule to satisfy the transition relation, we use a β rule and choose Y_j such that $\langle V_j \rangle b$ has a shortest derivation starting from the state s_{k+1}. Note that an appropriate Y_j must exist since Y_i set to true by a state s_k implies that $\sigma^{k..} \models \langle V_i \rangle b$. This fact can be easily proven by induction on the length of σ;

- Unless Y_i is forced to be true by the above rules, it is always interpreted as false.

It is easy to see that the above rules guarantee that σ obeys the transition relation. The termination condition is ensured by the first rule. The justice is also obvious as long as we can show that infinitely often all Y variables become false. Let N_k be number of steps of the longest shortest derivation sequence among al Y variables set to true at a state s_k. Clearly, after one step $N_{k+1} < N_k$. When N becomes 0 all Y variables are false.

3.5.2 A Tester for $\varphi = r$

We start the construction exactly the same way as we did for $\varphi = \langle r \rangle b$, in Section 3.5.1. Let $T_\varphi = \langle V_\varphi, \Theta_\varphi, R_\varphi, \mathcal{J}_\varphi, F_\varphi \rangle$ be the tester we wish to construct. Assume that x_φ is the output variable. Let $\mathcal{G} = \langle \mathcal{V}, \mathcal{T}, \mathcal{P}, \mathcal{S} \rangle$ be a grammar associated with r.

The tester T_φ is given by:

$$
T(r) : \left\{ \begin{array}{rl}
V_\varphi : & P \cup \{x_\varphi\} \cup \{X_1, \ldots, X_n, Y_1, \ldots, Y_n\} \\
\Theta_\varphi : & 1 \\
R_\varphi(V,\, V') : & \text{Each derivation } V_i \rightarrow \alpha_1 \mid \cdots \mid \alpha_m \mid \beta_1 V_1 \mid \cdots \mid \beta_n V_n \\
& \text{contributes to } R_\varphi \text{ the conjunct} \\
& X_i = \alpha_1 \vee \cdots \vee \alpha_m \vee (\beta_1 \wedge X_1') \vee \cdots \vee (\beta_n \wedge X_n') \\
& \text{and the conjunct} \\
& \alpha_1 \vee \cdots \vee \alpha_m \vee (\beta_1 \wedge Y_1') \vee \cdots \vee (\beta_n \wedge Y_n') \rightarrow Y_i \\
& \text{the output variable is constrained by the conjunct} \\
& x_\varphi = X_1 \\
\mathcal{J}_\varphi : & \{Y_1 \wedge \cdots \wedge Y_n, \quad X_1 = Y_1 \wedge \cdots \wedge X_n = Y_n\} \\
F_\varphi : & \text{Each derivation } V_i \rightarrow \alpha_1 \mid \cdots \mid \alpha_m \mid \beta_1 V_1 \mid \cdots \mid \beta_n V_n \\
& \text{contributes to } F \text{ the conjunct} \\
& X_i = \alpha_1 \vee \cdots \vee \alpha_m \vee \beta_1 \vee \cdots \vee \beta_n
\end{array} \right.
$$

The variables $\{X_1, \ldots, X_n, Y_1, \ldots, Y_n\}$ are expected to check that the rest of the sequence from now on has a prefix that does not violate SERE r. We follow a similar approach as for the tester $\varphi = \langle r \rangle b$. However, now we are more concerned with false values of the variables $X_1 \ldots X_n$. The duality comes from the fact that, now, we are trying to prevent postponing the violation of the formula r forever.

3.5.3 Handling *abort* operator

To handle *abort*, we rewrite a given formula to a semantically equivalent one not containing the *abort* operator. Let φ be a given formula. Without loss of generality, assume that $\varphi = \psi \ \mathit{abort}\ b$, where ψ is *abort*-free. An arbitrary formula can be processed by starting with an inner-most *abort* and removing them one by one.

For convenience, let *precedes* be the dual of *abort* such that $\phi \ \mathit{precedes}\ b \equiv \neg(\neg\phi \ \mathit{abort}\ b)$. Let f and g denote arbitrary PSL formulas; b, b_1, and b_2 denote boolean expressions; r denote a SERE. Let r_b be a SERE such that the formula $\phi = r_b$ is equivalent to $\phi = r \ \mathit{abort}\ b$. A simple algorithm to construct r_b, given r, is presented at the end of this section. We use the following equivalencies to rewrite φ:

- $b_1 \ \mathit{abort}\ b_2 \equiv b_1 \vee b_2$
- $(\neg f)\ \mathit{abort}\ b \equiv \neg(f \ \mathit{precedes}\ b)$
- $(f \wedge g)\ \mathit{abort}\ b \equiv (f \ \mathit{abort}\ b) \wedge (g \ \mathit{abort}\ b)$
- $(X!f)\ \mathit{abort}\ b \equiv b \vee X!(f \ \mathit{abort}\ b)$
- $(f \ U \ g)\ \mathit{abort}\ b \equiv (f \ \mathit{abort}\ b)\ U\ (g \ \mathit{abort}\ b)$
- $(\langle r \rangle f)\ \mathit{abort}\ b \equiv \langle r_b \rangle (f \ \mathit{abort}\ b)$
- $r \ \mathit{abort}\ b \equiv r_b$

- b_1 *precedes* $b_2 \equiv b_1 \wedge \neg b_2$
- $(\neg f)$ *precedes* $b \equiv \neg(f$ *abort* $b)$
- $(f \wedge g)$ *precedes* $b \equiv (f$ *precedes* $b) \wedge (g$ *precedes* $b)$
- $(X!f)$ *precedes* $b \equiv \neg b \wedge X!(f$ *precedes* $b)$
- $(f\ U\ g)$ *precedes* $b \equiv (f$ *precedes* $b)\ U\ (g$ *precedes* $b)$
- $(\langle r \rangle f)$ *precedes* $b \equiv \langle r\ \&\&\ \neg b[*]\rangle(f$ *precedes* $b)$
- r *precedes* $b \equiv r\ \&\&\ \neg b[*]$

Note that the size of the resulting formula is linear in the size of the original. In addition, while && is usually a very expensive operator, it is benign in our case since a grammar for $\neg b[*]$ has only one non-terminal, and can be completely eliminated from the rewriting rules.

Below are some addition equivalence rules that may be useful when deling with PSL formulas.

- $(f \vee g)$ *abort* $b \equiv (f \vee b) \wedge (g$ *abort* $b)$
- $(f \vee g)$ *precedes* $b \equiv (f \vee b) \wedge (g$ *abort* $b)$
- $(f\ \mathcal{W}\ g)$ *abort* $b \equiv (f$ *abort* $b)\ \mathcal{W}\ (g$ *abort* $b)$
- $(f\ \mathcal{W}\ g)$ *precedes* $b \equiv (f$ *precedes* $b)\ \mathcal{W}\ (g$ *abort* $b)$

We conclude the discussion of the *abort* operator by elaborating on the construction of r_b. Let $\mathcal{G}$ be a grammar associated with r. Our goal is to construct $\mathcal{G}' = \langle \mathcal{V}', \mathcal{T}', \mathcal{P}', \mathcal{S}' \rangle$ - a grammar associated with r_b.

- $\mathcal{V}' = \mathcal{V} \cup \{V_f\}$

- $\mathcal{T}' = \mathcal{T} \cup \{b\}$
- $\mathcal{P}' = \mathcal{P} \cup \{V_f \to \mathit{true}V_f, V_f \to \epsilon\} \cup \{V_i \to bV_f \mid V_i \in \mathcal{V}\}$
- $\mathcal{S}' = \mathcal{S}$

3.5.4 Complexity of the Construction

Theorem 3 *For every* PSL *formula* φ *of length* n*, there exists a tester with* $O(2^n)$ *variables. If we restrict SERE's to three traditional operators: concatenation* (;)*, union* (|)*, and Kleene closure* ([∗])*, the number of variables is linear in the size of* φ*.*

To justify the result, we can just count the fresh variables introduced at each step of the tester construction. There is only linear number of sub-formulas, so there is a linear number of output variables. The only other variables introduced are the ones that are used to handle SERE's. According to Theorem 1, the associated grammars contain at most $O(2^n)$ non-terminals ($O(n)$ - for the restricted case). We conclude by observing that testers for the formulas $\varphi = \langle r \rangle b$ and $\varphi = r$ introduce exactly two variables, X_i and Y_i, for each non-terminal V_i.

3.6 Using Testers for Model Checking

One of the main advantages of our construction is that all the steps, as well as the final result – the tester itself, can be represented symbolically. That is particularly handy if one is to use symbolic model checking [5]. Assume that the formula under consideration is φ, and $T_\varphi = \langle V_\varphi, \Theta_\varphi, R_\varphi, \mathcal{J}_\varphi, F_\varphi \rangle$ is the corresponding tester. Let JDS $\mathcal{D}$ represent the system we wish to model check.

We are going to use traditional automata theoretic approach based on synchronous composition, as in [5]. We perform the following steps:

- Compose $\mathcal{D}$ with T_φ to obtain $\mathcal{D} \mathbin{|||} T_\varphi$.
- Check if $\mathcal{D} \mathbin{|||} T_\varphi$ has a (fair) computation, such that $s_0[x_\varphi] = 0$. $\mathcal{D} \mathbin{|||} T_\varphi$ has such a computation iff $\mathcal{D}$ does not satisfy φ.

As you can see, a tester can be used anywhere instead of an automaton. Indeed, we can always obtain an automaton from a tester by restricting the initial state to interpret x_φ as *true*.

3.7 Run-time Monitoring with Testers

The problem of *run-time monitoring* can be described as follows. Assume a reactive system $\mathcal{D}$ and a PSL formula φ, which formalizes a property that $\mathcal{D}$ should satisfy. In order to test the conjecture that $\mathcal{D}$ satisfies φ, we construct a program M, to which we refer as a *monitor*, that observes individual behaviors of $\mathcal{D}$. Behaviors of $\mathcal{D}$ are fed to the monitor state by state. After observing the finite sequence $\sigma : s_0, \ldots, s_k$ for some $k \geq 0$, we expect the monitor to be able to answer a subset of the following questions:

1. Does σ satisfy the formula φ?
2. Is φ *negatively determined* by σ? That is, is it the case that $\sigma \cdot \eta \not\models \varphi$ for all finite or infinite completions η.
3. Is φ *positively determined* by σ? That is, is it the case that $\sigma \cdot \eta \models \varphi$ for all finite or infinite completions η?

4. Is φ σ*−monitorable*? That is, is it the case that there exists a finite η such that φ is positively or negatively determined by $\sigma \cdot \eta$. If $\mathcal{D}$ is expected to run forever then it is useless to continue monitoring after observing σ such that φ is not σ*−monitorable*.

Solving the above questions leads to a creation of an *optimal* monitor - a monitor that extracts as much information as possible from the observation σ. In particular, an optimal monitor detects a violation of the property as early as possible. Of course, a monitor can do better if we supply it with some implementation details of the system $\mathcal{D}$, which may allow to deduce a violation even earlier [53]. In the extreme case, when a monitor knows everything about $\mathcal{D}$ the monitoring problem is reduced to model checking.

3.7.1 Monitoring with Testers

Let $\mathcal{D} : \langle P, \Theta, R, \mathcal{J}, F\rangle$ be a reactive system with observable variables P, and let φ be a PSL formula over P, which validity with respect to $\mathcal{D}$ we wish to test. Assume that $T_\varphi = \langle V_\varphi, \Theta_\varphi, R_\varphi, \mathcal{J}_\varphi, F_\varphi\rangle$ is the tester for φ, where the variables $V_\varphi = P \cup A$ are partitioned into the variables of $\mathcal{D}$ and additional auxiliary variables A. Let x_φ be the distinguished output variable of the tester T.

For an assertion (state formula) α, we define the R_φ*-predecessor* and R_φ*-successor* of α by

$$R_\varphi \diamond \alpha \;=\; \exists V'_\varphi : R_\varphi(V_\varphi, V'_\varphi) \wedge \alpha' \qquad \text{and} \qquad \alpha \diamond R_\varphi \;=\; \mathit{unprime}(\exists V_\varphi : R_\varphi(V_\varphi, V'_\varphi) \wedge \alpha),$$

where *unprime* simply replaces all next state variables with current state variable.

Remember that the transition relation R_φ has two copies of each variable, one representing a current state and the other copy (a primed one) the next state.
Let $\sigma : s_0, s_1, \ldots, s_k$ be a finite observation produced by system $\mathcal{D}$. That is, a sequence of evaluations of the variables P. We define the *symbolic monitoring trace* $\mathcal{M} = \alpha_0, \alpha_1, \ldots, \alpha_k$ as the sequence of assertions given by

$$\alpha_0 = \Theta_\varphi \wedge x_\varphi \wedge (P{=}s_0), \text{ and } \alpha_{i+1} = (\alpha_i \diamond R_\varphi) \wedge (P{=}s_{i+1}) \text{ for all } i < k,$$

where $P = s$ stands for $\bigwedge_{v \in P} v = s[v]$.

Essentially, α_i represents a "current" state of the monitor, which is more precisely just a set of states of the tester T_φ. Whenever, the system makes a step from s_i to s_{i+1}, a monitor takes the corresponding step from α_i to α_{i+1} according to the transition relation R_φ and the interpretation of the propositions by the state s_{i+1}. The whole process can be described as, on the fly, synchronous, composition of the system and the tester, in which the later is determinized using classical subset construction. Note that we only need to worry about the existential non-determinism, A similar approach, but for alternating automata was also used for a so called breadth-first traversal in [24]. The monitoring sequence can be used to answer the first of the monitoring questions as stated by the following claim:

Claim 1 (Finitary satisfaction) *For a* PSL *formula φ, the finite sequence $\sigma : s_0, s_1, \ldots, s_k$ satisfies φ, i.e., $\sigma \vDash \varphi$, iff the formula $\alpha_k \;\wedge\; F_\varphi$ is satisfiable.*

The correctness of the claim results from the following observations. The tester T_φ can be interpreted as a non-deterministic automaton for acceptance of sequences satisfying φ if we insist that x_φ is *true* in the initial state. Furthermore, the assertion α_k represents all the automaton (tester) states which can be reached after reading the input σ. If any such evaluation is consistent with the assertion F_φ, which represents the set of final states, then this points to an accepting run of the automaton.

3.7.2 Deciding Negative Determinacy

Claim 1 has settled the first monitoring task. Next we consider one of the remaining tasks. Namely, we show how to decide whether, for a given σ, $\sigma \cdot \eta \not\models \varphi$ for all infinite or finite completions η.

In order to do this, we have to perform some offline calculations as a preparation. We generalize the notion of a single-step predecessor to an *eventual* predecessor by defining

$$R_\varphi^* \diamond \alpha = \alpha \vee R_\varphi \diamond \alpha \vee R_\varphi \diamond (R_\varphi \diamond \alpha) \vee \cdots$$

Consider the fix-point expression presented in Equation (3.1).

$$\mathit{feas} \quad = \quad [\mu X : (R_\varphi \diamond X) \vee F_\varphi] \quad \bigvee \quad [\nu Y : R_\varphi \diamond Y \ \wedge \ \bigwedge_{J \in \mathcal{J}} R_\varphi^* \diamond (Y \wedge J_\varphi)] \qquad (3.1)$$

The first expression captures all the states that have a path to a final state. The second expression captures a maximal set of tester states Y such that every non-final state $s \in Y$ has an Y-successor and, for every justice requirement J, s has a Y-path leading to some Y-state which also satisfies J. The following can be proven:

Claim 2 (Feasible states) *The set feas characterizes the set of all states which originate an uninitialized computation.*

Assuming that we have precomputed the assertion *feas*, the following claim tells us how to decide whether a finite observation σ is sufficient in order to negatively determine φ:

Claim 3 (Negative Determinacy) *The* PSL *formula* φ *is negatively determined by the finite observation* $\sigma = s_0, s_1, \ldots, s_k$ *iff* $\alpha_k \wedge \mathit{feas}$ *is unsatisfiable.*

The claim is justified by the observation that $\alpha_k \wedge \mathit{feas}$ being unsatisfiable means that there is no way to complete the finite observation σ into a finite or infinite observation which will satisfy φ.

3.7.3 Deciding Positive Determinacy

In order to decide positive determinacy, we need to monitor the incoming observations not only by assertion sequences which attempt to validate φ but also by an assertion sequence which attempts to refute φ. Consequently, we define the *negative symbolic monitoring trace* $\mathcal{M}^- = \beta_0, \beta_1, \ldots, \beta_k$ by

$$\beta_0 = \Theta_\varphi \wedge \neg x_\varphi \wedge (P{=}s_0), \text{ and } \beta_{i+1} = (\beta_i \diamond R_\varphi) \wedge (P{=}s_{i+1}) \text{ for all } i < k$$

Claim 4 (Positive Determinacy) *The* PSL *formula* φ *is positively determined by the finite observation* $\sigma = s_0, s_1, \ldots, s_k$ *iff* $\beta_k \wedge \mathit{feas}$ *is unsatisfiable.*

3.7.4 Detecting Non-Monitorable Prefixes

Unfortunately, not all properties can be effectively monitored. Consider a property $\square \diamondsuit p$, which is not σ-monitorable for any σ prefix. No useful information can be gained after observing a finite prefix if the property only depends on the things that must happen infinitely often. A good monitor should be able to detect such situations and alert the user. Next, we show how to decide whether φ is σ-monitorable, for a given σ.

Let $\mathcal{M} = \alpha_0, \alpha_1, \ldots, \alpha_k$ and $\mathcal{M}^- = \beta_0, \beta_1, \ldots, \beta_k$ be the positive and negative symbolic monitoring traces that correspond to σ. Let Γ represent a set of assertions. We define the *R_φ-successor* and *eventual R_φ-successor* of Γ by

$$\Gamma \Diamond R_\varphi \; = \; \{(\gamma \diamond R_\varphi) \wedge (P = s) \mid \gamma \in \Gamma, s \text{ is some state of the system } \mathcal{D}\}$$

and

$$\Gamma \Diamond R_\varphi^* = \Gamma \vee R_\varphi \Diamond \Gamma \vee R_\varphi \Diamond (R_\varphi \Diamond \Gamma) \vee \cdots$$

Claim 5 (Monitorability) *A* PSL *formula φ is $\sigma-$monitorable, where $\sigma = s_0, \ldots, s_k$, iff there exists an assertion γ such that either $\gamma \in (\alpha_k \Diamond R_\varphi^*)$ or $\gamma \in (\beta_k \Diamond R_\varphi^*)$, and $(\gamma \wedge feas)$ is unsatisfiable.*

The claim almost immediately follows from the definition of $\sigma-$monitorable properties, Claim 3, and Claim 4. Note that the algorithm can be very inefficient due to the double-exponential complexity. One way to cope with the problem is to consider each state in α_k and β_k individually. The idea is very similar to never-violate states introduced in [16]. A state of a Büchi automaton is called *never violate* if, on any input letter, there is a transition to another *never-violate* state. Similarly, we can define *never-satisfy* states and obtain a reasonable approximation to the problem of monitorability. Note that the complexity of this solution is exponential, which hopefully can be managed using BDD's. In addition, the never-violate and never-satisfy states can be pre-computed before the monitoring starts. However, it remains to be seen whether the approximation works well in practice.

3.8 Related Work

It is very interesting to compare our approach to the one suggested in [14], which uses alternating automata. We have already mentioned some high-level distinctions between testers and alternating automata in Section 3.2. However, the question remains about which construction is better. It turns out that both approaches yield very similar results, assuming universal non-determinism is removed from the alternating automata. Although that is a somewhat unexpected conclusion, it is not hard to justify it.

Without going into the details of algorithm described in [14], it is enough to mention that each state in the alternating automaton is essentially labeled with a sub-formula. To remove universal non-determinism, we follow classical subset construction. In particular, we assign a boolean variable x for each sub-formula φ to represent whether the corresponding state is in the subset. One can easily verify that x is nothing more but the output variable of the tester T_φ and follows the same transition relation.

To finish the partial determinization and define the final states in the new automata, the authors of [14] use the same trick with double representation as we do. At this step, the automata obtained after the subset construction is composed with itself via a cartesian product. This step is conceptually the same as introducing Y variables in the tester construction. However, we only introduce the extra variables when dealing with SERE's. For the LTL portion of the formula, the tester construction avoids the quadratic blow out associated with the cartesian product by essentially building a generalized Büchi with multiple acceptance sets (i.e., multiple justice requirements). If one is to insist on a single acceptance set, our approach would yield an automaton identical to the one obtained in [14]. Note that, for symbolic model checking, using a generalized Büchi automaton might be more efficient then the corresponding Büchi automaton.

While our approach may not necessarily yield a better automaton, it never performs worse, and there are several significant benefits. Since model checking is very expensive, we expect that, in practice, automata for commonly occurring subproperties will be hand-tuned. In such a case, it is more beneficial to work with testers since an alternating automaton requires an exponential blow-up due to universal non-determinism that cannot be locally optimized.

Another important advantage is that PSL testers can be used anywhere instead of LTL testers. For example, if one were to extend CTL* with PSL operators, our approach combined with [42] immediately gives a model checking algorithm for the new logic.

Chapter 4

Run-time Monitoring with Partially Specified Systems

There are two main sources of partially specified models. First, it is commonplace, that large software systems utilize third-party components. Recently the concept of Software as a Service (SaaS) is becoming more popular, where there is even a greater separation between internal parts of a system and external components. Due to the proprietary and engineering considerations in such a situation we may not always have a complete specification (i.e., all implementation details) of all parts comprising our system. Despite being beyond our full control, "off-the-shelf" components might still be attractive enough so that the designer of a new system may wish to use them. In order to do so safely, the designer must be able to deal with the possibility that these components may exhibit undesired or unanticipated behavior, which could potentially compromise the correctness and security of the whole system. Another important source of incomplete specifications is that we may intentionally over-approximate parts of our system to avoid state explosion problem associated with formal methods such as model checking. However, we can no longer directly apply model checking since it may lead to highly undesirable false positives and result in a perfectly good system being declared faulty. This problem, which is the result of *under-specification*, either intentional or not, is the central focus of this chapter

As a simple example of this phenomenon, consider an interface specification that guarantees "after input *query* q is received, output $r = \mathit{response}(q)$ is produced." The designer of the interface probably meant a stronger specification, "after q is received, nothing else is produced until r is produced." Assume that the later version is sufficient and necessary to ensure the correctness of the entire system consisting of the module and the interface. Applying formal methods, like model checking, would most likely fail since there is no algorithmic way to provide the model checker with the proper strengthening of the interface specification. Yet, under the assumption that interface specifications may be partial, there may exist a subset of the allowed behaviors that guarantees correctness, and one may still use the component, provided deviations of the interface from this "good" set of behaviors can be detected.

Assume that we are given:

- A finite-state *module* M, designed by our designer and accompanied by the full details of its implementation;
- An *interface specification* Φ_I for the external component interacting with the module M; and
- A *goal specification* Φ for the entire system which must be satisfied by the interaction between the module and the interface.

We view the module and interface as players in a 2-player game. At any stage in the computation we ask whether the interface has a strategy, consistent with its specification Φ_I, such that for any possible behavior of the module M, the behavior of the system resulting from the interaction of the module and the interface is guaranteed to satisfy the goal Φ. We use this successive game solving as a basis for run-time monitoring of the system, where we raise an alarm as soon as the system reaches a state from which the interface can no longer guarantee that the system behaves correctly (i.e., satisfies Φ).

A naive interpretation of the above description seems to imply that we solve a complete game after each move of either player. Such an implementation would make the process prohibitively expensive. Instead, we restrict our attention to games in which the winning condition is *universal liveness* – that is, closed under insertion and removal of arbitrary finite prefixes. For such games, it is sufficient to solve the game only once, and then just monitor the progress of the computation within the game structure. The single game solving process can be performed at compile-time, so there is very little to be done during the monitoring phase. In Section 4.2, we show that every game can be converted to a game with a universal liveness winning condition.

It is interesting to compare our methodology to more conventional run-time monitoring approach as the one described in [7, 21] or the one found in Chapter 3, Section 3.7. As far as we know, all the traditional run-time monitoring systems to a large degree ignore the implementation details of the program under consideration and concentrate on analyzing a specific behavior. Such monitoring systems work especially well if one is mostly interested in certifying that the observed execution trace is error-free and, possibly, collecting some statistical information. However, the conventional approaches are usually unacceptable if the main goal is to find faults in the design itself – not just in a particular computation. In contrast, in our framework, in addition to the run-time information, we are trying to use all the available implementation details. That ultimately leads to a higher precision since we monitor not only the current trace, but considerably more. The idea is similar to the *target enlargement* [67] and can become especially useful when debugging multi-threaded applications.

The chapter is organized as follows. In Section 4.1, how to associate a game with a given open system, its specification Φ, and the interface (environment) specification Φ_I. In Section 4.2, we show how to solve such games assuming all specifications Φ and Φ_I are expressed in PSL, and Section 4.3 describes the construction of the monitor. In Section 4.4, we discuss various aspects of our methodology. Finally, in Section 4.5, we present our conclusions and some possible future research directions.

4.1 Associating Games with JDS's

Given a JDS $M = (V_M, W, \Theta, \rho, \mathcal{J})$ that corresponds to some SPL module, a system specification Φ, and an interface specification Φ_I, we define a game $G = (S, A, \Gamma_1, \Gamma_2, \delta)$ between the module M and the interface as follows.

Let V be V_M augmented with a variable *turn* $\in \{1, 2\}$ that is not in V_M. Let S be the set of all V-states. The set A of G's actions is S itself. In the game G, player-1 (the module) can take any step that is allowed by M, non-deterministically setting *turn*, thus deciding whether or not it wishes to take another step. Player-2 (the interface) can set any of the variables that are not owned by M, but it has to let Player-1 take the next step. Formally:

- $\Gamma_1(s) = \{s' \mid \langle s, s' \rangle \models \rho\}$
- $\Gamma_2(s) = \{s' \mid \langle s, s' \rangle \models \Big(turn' = 1 \wedge \bigvee_{v \in V_M \setminus W} pres(V \setminus \{v, turn\})\Big)\}$
- $\delta(s, a_1, a_2) = a_{s[turn]}$. That is, δ selects a_1 as the next state iff $turn = 1$ in the current state s.

The *objective* of the game is defined by

$$\Psi = (\Phi \wedge \Phi_I) \vee \Diamond \Box (turn = 1) \vee \bigvee_{J \in \mathcal{J}} (\Diamond \Box \neg J).$$

Thus, the game is won by either meeting both Φ and Φ_I, violating one of the justice requirements, or preventing the interface from taking infinitely many steps. We force the module to give up its turn infinitely many times to preserve the semantics of interleaving.

4.2 Solving Games

Our approach to run-time monitoring is based on the observation that when the interface does not have a winning strategy a violation of the specification is unpreventable. Therefore the monitoring algorithm traces the interaction between module and interface and raises an alarm at the first time it detects that the interface no longer has a winning strategy. In Fig. 4.1, we present a naive implementation of this idea.

While the algorithm in Fig. 4.1 fully captures the spirit of our monitoring approach, it is computationally unacceptable. This is because it implies that we have to analyze the game each time afresh, with respect to an unbounded set of possible histories that may arise during computation.

- Check whether the initial state is winning for the interface (player-2) in game G. If it is not, then raise an alarm.
- In all subsequent steps, let h be the history observed since the beginning of the computation. If h is not a winning strategy for the interface, raise an alarm.

Figure 4.1: A naive algorithm for prevention maintenance

Note that the naive algorithm is feasible if we can *partition* the states of the game into *good* and *bad* ones so that, regardless of the history of the game, the interface can win for Ψ when the game is in a good state, and, similarly, the module can win for $\neg\Psi$ from a *bad* state. Formally, we define a state s to be *good* for player-i with respect to the objective Ψ if the following holds:

Every history ending in s is winning for player-i with respect to Ψ.

We define a state s to be *bad* for player-i with respect to the objective Ψ if:

State s is *good* for player-$(3-i)$ with respect to the objective $\neg\Psi$.

For example, states $\{s_0, s_1, s_2, s_3\}$ of the game of Fig. 2.1 are good for player-1 (w.r.t Ψ). These are the only states which are good for any of the players in this game. For example, the question whether state s_4 is winning depends on the path by which we reached s_4. If the path went through s_1, then s_4 is winning. If the path went through s_2, then s_4 is not a winning state. From here on, we apply the terms *good* and *bad* only with respect to player-2 (interface) and objective Ψ. A game G is called *partitionable* if every state s is either bad or good.

In case a game G is partitionable, it is easy to construct a monitor. First, we *solve* a game G by finding all good states. Note that in a partitionable game, the notions of a good state and a winning state coincide. Therefore, we can use the algorithm presented in [18] to find all player-2 winning states to solve a game. At run-time, we just need to make sure that the game doesn't enter a bad state and raise an alarm if it does.

Unfortunately, it is not always the case that a state s can be identified as good or bad.

Note that a game associated with a JDS can be easily represented as a turn based game with a Borel winning condition, which is known to be *determinate* [48, 17]. Determinacy guarantees that for each particular history h either player-1 can win for Ψ or player-2 can win for $\neg\Psi$, which is not true in general for the concurrent games described in Section 2.4 [19].

Therefore, the only reason why we cannot always partition the states is that for some states player-2 has a winning strategy for a history h, but not for some other history h'. This was the case, as shown above, for state s_4. Therefore, whenever the game reaches s_4 a monitor cannot immediately decide whether it should raise an alarm. We can, in principle, make a right decision by taking a closer look at the history of the game and using a variation of an algorithm that computes winning states. However, that would call for a fresh game analysis on each visit to s_4. Clearly, that would make monitoring too expensive.

To solve the problem, we characterize the games for which it is possible to partition the states into good and bad. A PSL formula Ω represents a *universal liveness* property if the following holds:

For every $\sigma_1 \in \Sigma^*$ and $\sigma_2 \in \Sigma^\omega$, $\sigma_1 \cdot \sigma_2 \models \Omega$ iff $\sigma_2 \models \Omega$.

Note that absolute liveness [3] satisfies only one direction of this definition, (i.e., if $\sigma_2 \models \Omega$ then $\sigma_1 \cdot \sigma_2 \models \Omega$). The notion of universal liveness has been considered in [59] under the name of *fairness*. Also, it should be stressed that universal liveness should not be confused with the concept of memoryless strategies (i.e strategies such that the choice of an action only depends on the last state of a game history). Indeed, a winning strategy for a game with a universal liveness winning condition may require memory and vice versa.

It can be shown that if the objective Ψ of game G is a universal liveness property, then G is a partitionable game. Intuitively, if there is a winning strategy for some history h, there must be one for any other history h', sharing the same last state. Therefore, there can be no state from which player-2 has a winning strategy only with respect to some history h but not with respect to some other history h' sharing the same last state.

In the remainder of this section, we will show how to transform an arbitrary game G to an equivalent game G' with an objective Ψ' that represents a universal liveness property. Essentially, we split undecided states like s_4 in Fig. 2.1 so that the new states can be identified as good or bad.

4.2.1 Converting the objective of a game expressed in PSL into universal liveness

Given a game structure $G = (S, A, \Gamma_1, \Gamma_2, \delta)$ and objective Ψ, we first build a tester that accepts Ψ as described in Chapter 3. Next we use the construction of [56] to build a total deterministic Rabin-chain automaton $R = (Q, q_0, \Delta, c)$ over S such that accepts Ψ.

The *composition of the game structure G with the Rabin chain automaton R* is the game $G \times R = (S', A', \Gamma'_1, \Gamma'_2, \delta')$, where:

- $S' = S \times Q$;
- $A' = A$;
- $\Gamma'_i((s, q)) = \Gamma_i(s)$ for $i = 1, 2$;
- $\delta'((s, q), a_1, a_2) = (\delta(s, a_1, a_2), \Delta(q, s))$.

It is straightforward to convert the acceptance condition of R into an LTL objective Ψ' for the game $G' = G \times R$. Indeed, we can define Ψ' as:

$$\Psi' = \bigvee_{i=0}^{k-1} \big(\Box \Diamond (c = 2i) \wedge \Diamond \Box (c \leq 2i)\big),$$

where c is interpreted in a state (s, q) as $c(q)$. Since the formula Ψ' consists of a boolean combination of formulas of the form $\Box \Diamond p$ and $\Diamond \Box p$ for assertions p, it is easy to show that Ψ' represents a universal liveness property. We can solve game G' using the algorithm from [18].

Example 5 Consider the game structure in Fig. 2.1. The automaton for the objective Ψ is given in Fig. 2.2, and the composed game is outlined in Fig. 4.2.

The only good states in the composed game are (s_4, q_0) and (s_5, q_0). All other states are bad. Thus, as desired, this composed game with universal liveness objective is partitionable.

4.2.2 Deterministic Büchi Specifications

The construction above is quite expensive. The size of the resulting game is doubly exponential in the length of the original objective Ψ. In addition, the size of the new objective Ψ' is exponential in the length of Ψ. An important and frequently occurring

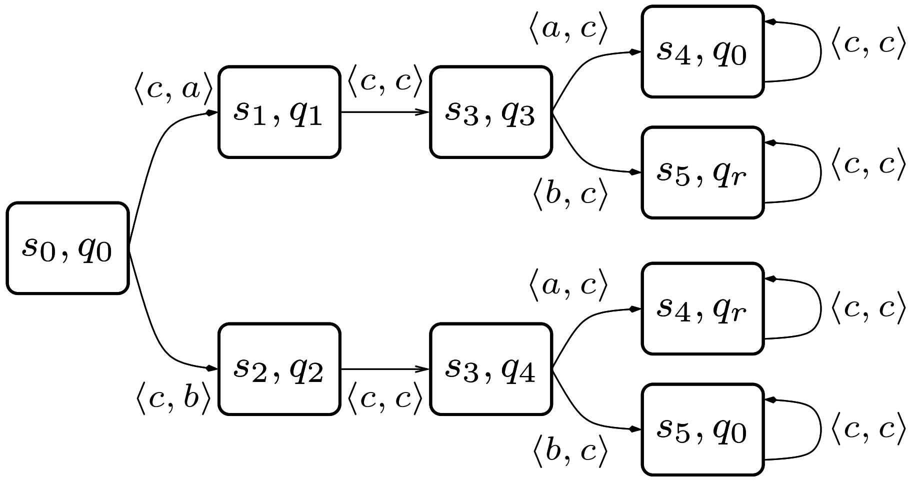

Figure 4.2: The composed game

special case is when the specification $\Phi \wedge \Phi_I$ can be represented as a deterministic Büchi automaton. In this case, we can skip the expensive determinization step. We present an efficient algorithm for monitoring of such special cases.

As defined in Section 4.1, the objective of our game G is defined by

$$\Psi = (\Phi \wedge \Phi_I) \vee \Diamond \Box (\mathit{turn} = 1) \vee \bigvee_{J \in \mathcal{J}} (\Diamond \Box \neg J).$$

Recall that to solve a game for the purpose of run-time monitoring, we need to use a partitionable game. In Section 4.2.1, we have presented a general methodology for transforming an arbitrary game into a game with universal liveness objective. Here we consider the special case that $(\Phi \wedge \Phi_I)$ can be represented by a deterministic Büchi automaton. We proceed as follows:

- Build a deterministic Büchi automaton B that accepts $(\Phi \wedge \Phi_I)$. Let $\mathcal{F}$ represent the accepting set.

- Compose the automaton B with the game G to obtain $G' = G \times B$ as in

Section 4.2.1. Let the objective of the resulting game G' be defined by

$$\Box \Diamond \mathcal{F} \vee \Diamond \Box (turn = 1) \vee \bigvee_{J \in \mathcal{J}} (\Diamond \Box \neg J).$$

- Consider an LTL objective Φ in the form:

$$\Box \Diamond p \vee \bigvee_{i=1}^{n} (\Diamond \Box r_i),$$

where $p, r_1, \ldots, r_n$ are assertions. Clearly, the objective of G' has such a form. Let *Win* be a set of player-2 winning states for Φ. Following [41], we can compute *Win* as follows:

$$Win = \nu Z.\mu Y.\Big(\bigvee_{i=1}^{n} \nu X.(p \wedge \circledcirc Z) \vee \circledcirc Y \vee (r_i \wedge \circledcirc X)\Big),$$

where

$$[[\circledcirc f]]_G^e = \{s \in S \mid \exists b \in \Gamma_2(s).\forall a \in \Gamma_1(s).\ \delta(s, a, b) \in [[f]]_G^e\}.$$

That is, $[[\circledcirc f]]_G^e$ characterizes the set of states from which player-2 can force the game to move to a state belonging to $[[f]]_G^e$ in one step. For example, applying $\circledcirc$ to the set $\{s_1, s_2\}$, in Example 1, yields $\{s_0\}$.

The above formula can be evaluated symbolically, requiring at most $|S|^2$ steps, in spite of the fact that it has three alternating fix-point operators [45].

4.3 Feasible Interface Monitoring

Assume a module M, a specification Φ, and an interface specification Φ_I. A finite prefix σ of M-states is *safe with respect to* M*,* Φ*, and* Φ_I if there exists a JDS I^* (a possible interface), such that

- σ is a prefix of some computation of $M \parallel I^*$;
- For every computation σ' of $M \parallel I^*$ that has σ for a prefix, $\sigma' \models \Phi \wedge \Phi_I$.

The JDS I^* can be viewed as a concrete implementation of a winning strategy of player-2 (the interface). In fact, it can be shown that every such interface induces a winning strategy applicable after observing σ.

Based on the discussion in the preceding sections, we can now formulate a more efficient version of run-time monitoring process, in which we analyze the game only once — prior to the beginning of the monitored production run.

Using the methods of the previous section, we construct a partitionable game G which represents the possible interaction between the module and the interface, while assessing whether this interaction satisfies the conjunction $\Phi \wedge \Phi_I$ and the relevant fairness requirements of the module and proper interleaving between module and interface. Since G is partitionable, we will use the terms "winning" and "good" interchangeably. Let `Init` be the set of states that satisfy the initial condition Θ of M and any initial conditions induced by any automaton that may have been combined into G. A sketch of a feasible monitoring algorithm is presented in Fig. 4.3.

The following theorems state the soundness of Algorithm MON.

Theorem 4 *If* `Init` $\not\subseteq$ `Win`*, then* M *is incompatible with* $\Phi \wedge \Phi_I$*. That is, there exists no interface* I^* *such that* $M \parallel I^* \models \Phi \wedge \Phi_I$*.*

- Prior to the monitored production run, analyze the game G, computing the set `Win` of states which are winning for the interface. If `Init` $\not\subseteq$ `Win`, then raise an alarm.
- Start the production run. For every observed finite prefix σ of M-states, let h be the corresponding history of G, induced by σ. Let s be the last state of h. If $s \notin$ `Win`, then raise an alarm.

Figure 4.3: A feasible algorithm MON for prevention maintenance

Theorem 5 *Algorithm* MON *in Fig. 4.3 alerts only after observing an unsafe history.*

The proof of both theorems is based on the observation that if a state s, reachable by history h, is not winning for the interface, then there exists no interface I^* which, when run in parallel with M, can guarantee that all continuations of h do not violate $\Phi \wedge \Phi_I$.

If there existed such an interface, we could derive from it a strategy which is winning for the interface, contradicting the fact that the monitor observed a state which is bad for the interface. Thus, if an alarm is raised, there is no way to prevent a computation which violates Ψ.

More formally, we need to show that every infinite history h violating Ψ induced by a run of a game is an open computation of M violating $\Phi \wedge \Phi_I$, where M infinitely often gives up its turn to the environment. Indeed, consider a history h which violates the objective $(\Phi \wedge \Phi_I) \vee \Diamond \Box (turn = 1) \vee \bigvee_{J \in \mathcal{J}} (\Diamond \Box \neg J)$. Violation of $\bigvee_{J \in \mathcal{J}} (\Diamond \Box \neg J)$ guarantees that h satisfies all the fairness requirement of M. Thus, h is an open computation of M violating $\Phi \wedge \Phi_I$. Violation of the conjunct $\Diamond \Box (turn = 1)$ guarantees that h contains infinitely many interface steps.

Note that since in the absence of design faults all prefixes are safe, Theorem 5 implies that if an alert is generated, there is a bug in the system. Since we are interested in finding faults, the above statement signifies soundness of our method. In contrast, in model checking, the above statement usually means completeness since the goal is to prove program correctness.

Example 6 Consider the JDS corresponding to the SPL module in Fig. 4.4(a). Assume the specification is $\Phi : \Box \Diamond at\_\ell_0 \wedge \Box \Diamond \mathit{flag}_2$ and the interface specification is the trivial $\Phi_I : \mathsf{true}$. In Fig. 4.4(b) we present an SPL program for an interface I^* such that $M \parallel I^* \models \Phi \wedge \Phi_I$. Therefore, the module is compatible with $\Phi \wedge \Phi_I$. As was mentioned before, we can view I^* as a concrete realization of a winning strategy for the interface. Consider a state $at\_\ell_2 \wedge \mathit{flag}_1 \wedge \mathit{flag}_2$. It is easy to verify that it is a bad state, assuming it is the module's turn. Consequently, our monitor will raise an alarm when such state is reached. The alarm is really an *early* warning. Although the violation is unpreventable, it would take infinitely many steps to confirm the violation.

Besides catching the problem early on, there is another significant advantage of our approach – the fact that we were able to identify the problem at all. There is no way to identify a violation of a liveness property, like $\Phi : \Box \Diamond at\_\ell_0 \wedge \Box \Diamond \mathit{flag}_2$, using traditional run-time monitoring unless the program terminates. Of course, we cannot always catch a violation of liveness either, but, sometimes, we can catch an eventual violation, as we did in this example. Moreover, let's modify Φ to be $\Box \neg at\_\ell_4$. Even in that case, there is no guarantee that a user would be alerted when traditional run-time monitoring is employed since there is still a chance that the specification will hold. Indeed, the module can give up the turn and the interface can reset the $flag_2$ to false, which would restore the game into a good state.

```
        shared  boolean  $flag_1 = 0$
        shared  boolean  $flag_2 = 0$
        local   boolean  $error = 0$
$\ell_0$ :  while $\neg error$
            [ $\ell_1$ :  $flag_1 := 0$
              $\ell_2$ :  await $flag_1$
              $\ell_3$ :  if $flag_2$ then $error := 1$ ]
$\ell_4$ :
```

(a) Module M

```
        shared  boolean  $flag_1 = 0$
        shared  boolean  $flag_2 = 0$
$m_0$ :  loop forever do
            [ $m_1$ :  await $\neg flag_1$
              $m_2$ :  $flag_2 := 1$
              $m_3$ :  $flag_2 := 0$
              $m_4$ :  $flag_1 := 1$ ]
$m_5$ :
```

(b) Interface I^*

Figure 4.4: Module and a possible interface for Example 6

4.4 Discussion

Soundness and Completeness of the Method Our method is sound, meaning that whenever MON raises an alert, there is a bug in the implementation of the module and/or external components. However, as we mentioned before, an execution trace that generates an alert is not actually bound to violate specification. Nevertheless, an alert should be treated with the same amount of respect as a negative result of model checking. Of course, in practise, it might be desirable to distinguish when there is just a bug in the program or when a bug will cause the currently observed execution to violate specification. To accomplish this, we construct a game as before but let the interface make choices not only for itself but also for the module. If the interface cannot win even under these relaxed conditions, then a violation is guaranteed to occur whenever an alert is generated.

We also need to mention incompleteness of algorithm MON, which in our case means that the existence of a winning strategy for the interface does not necessarily imply that this strategy can be implemented by an SPL program. It is mainly due to the following two reasons. First, we allow the interface to observe all the variables of M, even the local ones. In addition, the interface has access to the complete history of a computation. The practical effect of incompleteness is that a monitor may not spot a bug as early as it could have done; sometimes it may not deduce a bug at all. However, it should be stressed that as soon as a computation violates the specification an alert will be generated. Therefore, in this respect, MON is always better then traditional run-time monitoring tools.

Note that incompleteness of algorithm MON should not be confused with the following fact – the absence of an alarm does not guarantee that the computation actually satisfies the objective if we let the game continue forever. Just because the game stays within good states, the system may not be getting any closer to satisfying its liveness properties. Unfortunately, any sound run-time monitoring system has the above characteristic.

State Explosion Consider the formula for computing the winning states at the end of Section 4.2.2. The amount of work needed to compute the set *Win* is comparable to model checking the module in the presence of fairness. Both require handling at least two alternating fix-point operators. Consequently, both suffer from state explosion problem. Fortunately, the parallel with model checking does not stop here, and we can apply some popular remedies used for model checking. In particular, one can abstract the module before applying run-time monitoring. To preserve the soundness of the method, we can allow the interface to resolve all the non-deterministic choices caused by abstraction. Of course, this may adversely effect completeness.

Interestingly enough, one of the reasons to use run-time monitoring in the first place may be to cope with the complexity of model checking. Indeed, we can label parts of the program that are hard to deal with as external components and apply run-time monitoring. Since we allow specifications given as Rabin-chain automaton, one can abstract to any degree that is necessary. In the extreme case when specification of the external components matches their implementation, run-time monitoring is degraded to regular model checking.

4.5 Our Contribution and Related Work

Game-theoretic formalisms have been widely applied for solving various verification related problems. For example, a problem of verification and synthesis of open systems is studied in [46] and is closely related to our work. However, as far as we know, we are the first to utilize game-theoretic approach for run-time monitoring. In addition, we have identified and proposed a solution for a problem of solving games in the presence of dynamic information.

We have already mentioned some differences between our work and "traditional" approaches to run-time monitoring. Most importantly, we are pursuing different goals: trace monitoring vs fault discovering. Therefore, we largely view our work as being orthogonal to the existing methods. However, even if we concentrate on a specific behavior like traditional monitors do, our approach offers several advantages. First, we are able to deal with liveness properties, raising an alarm when a violation of a liveness property can no longer be prevented with absolute assurance by the interface. Furthermore, in most cases, we can detect violations long before they actually happen. One such case is when an individual thread in a multi-threaded application performs a sequence of local computations, which are usually deterministic. If something is about to go wrong during this time, an alert is generated before any actual computations takes place. This information can be helpful in a number of ways, for example, it might be prudent to increase the level of logging to facilitate the debugging/recovery process later on. In addition, one may even attempt to repair the faulty application as was discussed in [39].

Chapter 5

Range Analysis

Model checking [5] is a promising verification technique, which has been used successfully in practice to verify complex circuit designs and communication protocols. However, model checking suffers from the *state explosion* problem, i.e. the number of states to explore grows exponentially with the number of state elements. This problem is further exacerbated in the context of software verification, where variables are typically modeled as multi-bit state vectors and arrays of variables are common.

In this chapter, we directly address this problem by bounding the number of bits needed to represent program variables. We statically determine possible ranges for values of variables in programs and use this information to extract smaller verification models. The use of this information greatly improves the performance of back-end model checking or static analysis techniques, based on use of BDDs or SAT solvers.

Our main method formulates the range analysis problem as a system of inequality constraints between symbolic bound polynomials. It then reduces the constraint to a Mixed Linear Programming (MLP) problem. A second approach called bounded range analysis was introduced for bounded depth analyses as is done in Bounded

Model Checking (BMC) [8]. Both methods have been implemented in F-Soft verification platform [33].

Outline. We have presented some background information on our software modeling approach, centered around a Control Flow Graph (CFG) representation of a software program in Chapter 2, Section 2.6. In Section 5.1 we present the generation of the range analysis constraint system in terms of the CFG representation. Section 5.2 discusses the analysis of the constraint system using an MLP solver. Section 5.3 presents our new bounded range analysis technique. We present experimental results with our prototype implementation in Section 5.4, and conclude this chapter with some remarks.

5.1 Constraint System Generation

The first step in range analysis is the generation of a symbolic constraint system, which is then analyzed by a linear program solver as described in Sec. 5.2. This section describes the generation of this constraint system.

First, we describe the basic outline; a detailed description will follow. For each program variable v, we need a bound on the number of bits needed to represent the value of v in all possible executions. We actually compute something more general: for each variable v and each program location *loc*, we compute the lower and upper bounds of v at *loc* in terms of the arguments of the main function [55]. That is, we derive explicit linear expressions that let us bound the possible values of v at *loc* for any given values of the main function arguments. Currently, we use only some of this information for improving model checking. Specifically, for a variable v whose bounds at all program points do not depend on main function arguments, we assign to v the smallest range that contains the ranges of v computed at *all* locations. From this range, we derive the number of bits needed to represent v. For variables whose bounds at some locations depend on the values of main function arguments, we cannot compute a meaningful bound because in most of our model-checking analyses the values of main function arguments are assumed to be arbitrary. If we only need to model check the program for specific ranges of main function arguments, then the bounds on variables whose ranges depend on main function arguments can be used to bound the bitwidths of these variables.

5.1.1 Symbolic Bounds

For each block B_i in the CFG of procedure f we define two locations: pre_i representing the start of block B_i and $post_i$ representing the end of B_i. The set of local integer variables (including pointer variables) of procedure f is denoted by V_f. We use v_{loc} to denote the value of the variable v at program location loc. $P_f \subseteq V_f$ is a set of formal parameters of procedure f that are defined to be integers. For each formal parameter $p \in P_f$ we use p_0 to symbolically represent the value of the actual parameter that corresponds to p. Similar to [55], we focus the rest of our discussion on the case where the values of the actual parameters are positive.

It should be noted that the set P_f of formal parameters is typically empty in our analyses, since each analysis treats one function as *main* with unconstrained argument values; computing ranges in terms of these unconstrained arguments is not useful for us, and so we only look for ranges that are independent of the argument values. Additionally, since we consider one entry function f we only need to consider one set V_f and one set P_f.

For each variable $v \in V_f$ and location loc, let L^v_{loc} and U^v_{loc} represent lower and upper bounds, respectively, of the value of v at loc. We set L^v_{loc} and U^v_{loc} to be linear combinations of the parameters of f with unknown rational coefficients, which are formally defined as follows:

$$\begin{aligned} L^v_{loc} &= C^L + \sum_{p \in P_f} C^L_p \cdot p_0, \\ U^v_{loc} &= C^U + \sum_{p \in P_f} C^U_p \cdot p_0. \end{aligned}$$

In order to obtain the lower and upper bounds of each variable, we generate the constraint system by considering following three types of constraints.

Initialization constraints. We generate initialization constraints for location pre_0 that represents the beginning of the initial block B_0. For each $p \in P_f$ we require that $L^p_{pre_0} = U^p_{pre_0} = p_0$. For each $v \in V_f \backslash P_f$ we require that $L^v_{pre_0} = -\infty$ and $U^p_{pre_0} = +\infty$. The values of "$-\infty$" and "$+\infty$ correspond to the most conservative lower and upper bounds of the particular integer data type used. For unsigned integers, $-\infty \equiv 0$ and $+\infty \equiv 2^{32} - 1$; for signed integers, $-\infty \equiv -2^{31}$ and $+\infty \equiv 2^{31} - 1$.

Assignment constraints. Assignment constraints define the bounds after execution of the expressions in a block. For each assignment, we update the bounds at the corresponding block of the variable on the left hand side with the bounds of the expression found on the right hand side. We define $l(e, loc)$ to represent the lower bound of an expression e at location loc. We compute $l(e, loc)$ for a constant c, a variable v and expressions e, e_1 and e_2 as follows:

$$\begin{array}{lcl} l(c,\, loc) & = & c \\ l(v,\, loc) & = & L^v_{loc} \\ l(e_1 + e_2,\, loc) & = & l(e_1,\, loc) + l(e_2,\, loc) \\ l(e_1 - e_2,\, loc) & = & l(e_1,\, loc) - u(e_2,\, loc) \\ l(c \cdot e,\, loc) & = & \begin{cases} c \cdot l(e,\, loc) & c \geq 0 \\ c \cdot u(e,\, loc) & c < 0 \end{cases} \end{array}$$

Whenever we cannot compute a bound, we let $l(e,\, loc) = -\infty$.

Similarly, we can define $u(e, loc)$ for upper symbolic bounds of expressions.

For an assignment within block B_i of the form $v = e$, where $v \in V_f$, we generate the following assignment constraint (note that a variable can be assigned at most once in a basic block after simplication):

$$L^v_{post_i} = l(e, pre_i) \wedge U^v_{post_i} = u(e, pre_i).$$

In case a variable v is not reassigned in block B_i, we generate the following constraint:

$$L^v_{post_i} = L^v_{pre_i} \bigwedge U^v_{post_i} = U^v_{pre_i}.$$

Propagation constraints. If a transition can be taken from a block B_i to some block B_j, a range of a variable v at the beginning of B_j must include all possible values the variable v can have just before such a transition. Formally we add the following constraint to the constraint system if there is a transistion from B_i to B_j:

$$L^v_{pre_j} \leq L^v_{post_i} \bigwedge U^v_{pre_j} \geq U^v_{post_i}.$$

5.1.2 Handling of Conditionals

The constraint system as defined in the previous section is comprehensive enough to ensure soundness of the bounds. However, additional information implied by conditionals (guards) in the CFG may further minimize the resulting ranges. For example, consider that the range of a variable v before a conditional is $v \in [0, 100]$, but the condition guarding the transition to a new block is $v \geq 20$. If there is no other incoming edge to the new block, then the lower bound for v in the new block can be safely assumed to be 20.

In general, consider a transition from block B_i to B_j and a guard of the form $v \geq e$, where $v \in V_f$. Assume that $L^v_{pre_j} \leq l(e, post_i)$ is satisfied.

Then, whenever we can make a transition from B_i to B_j we are guaranteed that the lower bound of v at the beginning of B_j is less or equal to the value of v at the end of B_i at the time of the transition. So, we can relax the corresponding flow constraint to:

$$L^v_{pre_j} \leq L^v_{post_i} \bigvee L^v_{pre_j} \leq l(e, post_i).$$

Often we can omit the flow constraint altogether. However, since we do not know a priori the relationship between $L^v_{post_i}$ and $l(e, post_i)$ we introduce a disjunction that results in higher precision. Other comparison operators can be handled in a similar fashion.

```
void foo(int N){
    int i = 0;
    while ( i ≤ N) i + +;
}
```

Figure 5.1: A Sample Program

However, the addition of disjunctions into an otherwise purely conjunctive constraint system presents a challenge to the approach advocated here. We use LP solvers to perform an efficient analysis of the constraint system. Allowing disjunctions prevents us from using pure LP solvers where all variables are rational (since disjunctions are not linear). Therefore, the original work suggested in [55] does not allow disjunctions. We discuss our disjunction handling in Sec. 5.2.

As an example, consider the program presented in Fig. 5.1. The corresponding CFG and symbolic constraint system are shown in Fig. 5.2. We omit the constraints for lower bounds in order not to clutter the figure.

5.1.3 Objective Function

Since we are interested in the most precise range information, we add an objective function to the LP problem. It minimizes the total number of values to be considered: $\sum_{v \in V_f} \sum_{B_i \in R_v} \mid U^v_{pre_i} - L^v_{pre_i} \mid$, where $R_v = \{B_i \in B \mid v \text{ is read in block } B_i\}$.

5.1.4 Constraint System Decomposition

Intuitively, it is clear that some bounds are independent of some other bounds. To formalize this notion we follow [55] and introduce a dependency graph of bounds. The nodes in the graph represent bounds. For a block B_i and a variable v there are

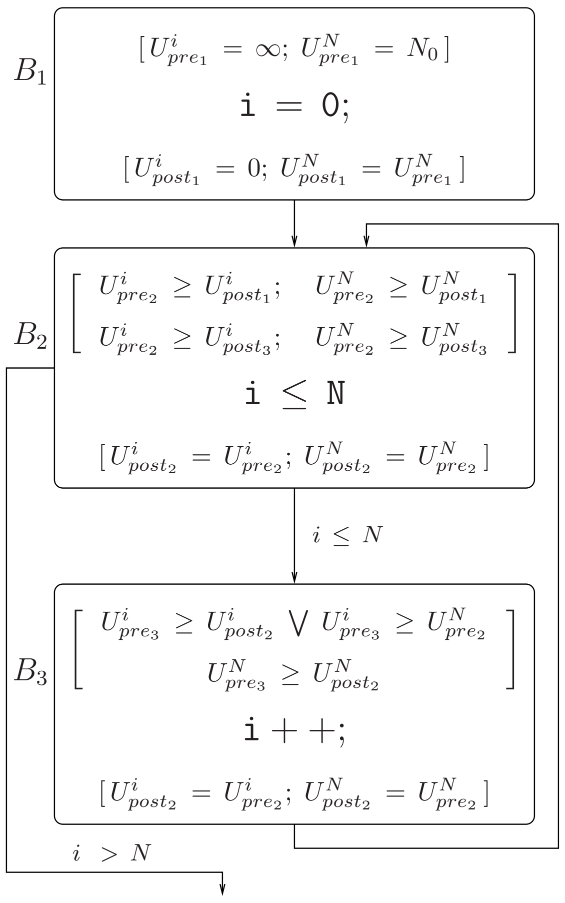
B_1
$[U^i_{pre_1} = \infty;\ U^N_{pre_1} = N_0]$
$\mathtt{i} = 0;$
$[U^i_{post_1} = 0;\ U^N_{post_1} = U^N_{pre_1}]$
B_2
$U^i_{pre_2} \geq U^i_{post_1};\quad U^N_{pre_2} \geq U^N_{post_1}$
$U^i_{pre_2} \geq U^i_{post_3};\quad U^N_{pre_2} \geq U^N_{post_3}$
$\mathtt{i} \leq \mathtt{N}$
$[U^i_{post_2} = U^i_{pre_2};\ U^N_{post_2} = U^N_{pre_2}]$
$i \leq N$
B_3
$U^i_{pre_3} \geq U^i_{post_2} \bigvee U^i_{pre_3} \geq U^N_{pre_2}$
$U^N_{pre_3} \geq U^N_{post_2}$
$\mathtt{i}++;$
$[U^i_{post_2} = U^i_{pre_2};\ U^N_{post_2} = U^N_{pre_2}]$
$i > N$

Figure 5.2: Symbolic Constraint System

exactly 4 nodes in the graph corresponding to $L^v_{pre_i}$, $U^v_{pre_i}$, $L^v_{post_i}$, and $U^v_{post_i}$. For every generated constraint there is an edge from the node that represents the bound on the left hand side to any node that represents a bound on the right hand side. We then decompose the graph to strongly connected components and process each component separately in reverse topological order. Decomposition prevents the propagation of unboundedness between unrelated variables. This allows us to find tight bounds for many variables.

5.2 Analyzing a Constraint System

Although the generated constraint system resembles an LP problem, several important distinctions prevent the direct usage of LP solvers. First, lower and upper bounds in the generated constraint system are linear expressions with not only unknown variables, but also unknown rational coefficients. Second, the generated constraint system has both conjunction and disjunction as Boolean connectives, while LP solvers only support conjunction. In order to use a LP solver, we must simply the generated constraint system.

5.2.1 From Symbolic Constraints to Linear Inequalities

Consider a constraint of the form $L^v_{loc'} \leq L^v_{loc}$, where $L^v_{loc} = C + \sum_{p \in P_f} C_p \cdot p_0$ and $L^v_{loc'} = C' + \sum_{p \in P_f} C'_p \cdot p_0$. We generate the following linear inequality constraint that can be submitted to the LP solver:

$$C' \leq C \wedge \Big(\bigwedge_{p \in P_f} C'_p \leq C_p \Big).$$

Assuming positivity of parameters, the new constraint is stronger than the original one, thus preserving soundness of the bounds. Other constraints can be handled similarly. When we cannot assume positivity of parameters, we need to perform an inefficient case split for the possible combinations of positive and negative parameters.

5.2.2 From Symbolic Objective Functions to LP Objective Functions

Similarly, we convert a symbolic objective function of the constraint system into a linear objective function. Assuming that $L^v_{pre_i} = X^v_{pre_i} + \sum_{p \in P_f} X^v_{pre_i,p} \cdot p_0$ and $U^v_{pre_i} = Y^v_{pre_i} + \sum_{p \in P_f} Y^v_{pre_i,p} \cdot p_0$, we rewrite the objective function as

$$\begin{aligned}&\textstyle\sum_{v \in V_f} \sum_{B_i \in R_v} \mid (Y^v_{pre_i} - X^v_{pre_i}) + \\ &\textstyle\sum_{p \in P_f} (Y^v_{pre_i,p} - X^v_{pre_i,p}) \mid .\end{aligned}$$

5.2.3 Handling Disjunctions

The addition of disjunctions into an otherwise purely conjunctive constraint system presents a challenge to the approach using LP solvers. If the user chooses to consider disjunctions, our tool uses an approach based on encoding disjunctions via integer variables. Several heuristics are used to reduce the number of disjunctions. We describe our approach using a small example. Consider the following constraint: $L^v_{pre_j} \leq L^v_{post_i} \bigvee L^v_{pre_j} \leq l(e, post_i)$. We introduce two new binary variables D_1 and D_2, and M denotes a large positive number. Our original constraint is then replaced with the following constraint:

$$\begin{aligned}&\big(D_1 + D_2 \leq 1\big) \bigwedge \big(L^v_{pre_j} - M \cdot D_1 \leq L^v_{post_i}\big) \\ &\bigwedge \big(L^v_{pre_j} - M \cdot D_2 \leq l(e, post_i)\big).\end{aligned}$$

The new constraint is stronger than the original one, and the two constraints are actually equivalent if M is sufficiently large. Note that the new variables D_1 and D_2 are the only variables that need to be pure integer variables, while all others can have rational values. For problems with small numbers of integer variables, performance of MLP solvers is comparable to the performance of LP solvers.

In the following we briefly describe some of the heuristics we employ to resolve some disjunctions before we invoke the appropriate LP solver for the resulting constraint system.

- *Drop a constraint if some bound on the right hand side is close to M.* For several practical reasons, mostly due to lack of necessary precision of floating-point numbers, we may not be able to set M as high as we wish. If we were not to drop such disjunctive constraints, we may actually treat the disjunctive constraint as a conjunctive constraint instead. Therefore, we resolve this problem by dropping a constraint from the disjunction.

- *Drop a constraint if some bound on the right hand side has not yet been determined.* As in the previous case the unknown bound value might be close to M and we would have the same issue that was described in the previous heuristic. Furthermore, if the unknown value cannot be bounded, the whole constraint system would be declared infeasible and thus unbounded.

- *Prefer to satisfy the flow constraint.* It is clear, that we cannot drop both sides of a disjunction. That is, if the aforementioned rules require that both constraints of a disjunction be removed, we need to keep at least one of the two in the resulting constraint system. In such a case, we prefer to leave the flow constraint in the constraint system, since it refers to one program variable only.

5.3 Bounded Range Analysis

Recently, there has been a growing interest in utilizing bounded model checking for program verification [13, 31]. We propose the idea of *bounded range analysis*, which computes ranges by exploiting the fact that the range information, if used only in a bounded model checking run of depth k, does not have to be sound for all computations of the program, but only for traces up to length k. By concentrating on a bounded length trace only, we are able to find tight bounds on many program variables that cannot be bounded using the technique described earlier. Such an approach can also support non-linear functions. As an example, consider the following code:

$$\textbf{int}\ i = 0, j = \mathit{readInput}(); \quad \textbf{while}(i < j * j)\ \{i++;\}$$

If one were to consider all possible traces and j is not bounded, then the upper bound for i would have to be declared unbounded. However, if we are only concerned with the traces up to k steps, it is safe to conclude that the value of i will always be in the range from 0 to k.

A straightforward way to compute such ranges is to perform a BFS on the control flow graph with depth limit set to k which updates the lower and upper bounds for the individual basic blocks. Although this approach results in very precise ranges, it is not efficient for large k. We propose the following algorithm that can be easily implemented on top of the constraint based approach described earlier. For a fixed number of steps (depth), its runtime is quadratic in terms of the code size.

```
Intialize all bounds to the least conservative values;
**for**($i = 0$; $i < \#steps$; $i++$)
  **foreach** basic block $B_j$
    **foreach** variable $v$, $v \in V_f$
      update $L^v_{pre_j}$, $U^v_{pre_j}$, $L^v_{post_j}$, and $U^v_{post_j}$
      using constraints
```

This algorithm can be further improved in several ways, in particular to support non-linear functions.

- *Support for non-linear functions.* In case a function does not have any parameters, we can easily extend the algorithm to support many important non-linear functions. The restriction on the presence of parameters is not severe. Expressing bounds as a linear combination of parameters is mostly useful for inter-procedural analysis. Since F-Soft inlines all function calls, parameters of called functions can be ignored. Consider an assignment $y = x^2$ in a block B_i. The following rules can be used to update $L^y_{post_i}$ and $U^y_{post_i}$: $L^y_{post_i} = 0$ and $U^y_{post_i} = \max(\mid L^x_{pre_i} \mid, \mid U^x_{pre_i} \mid)^2$.

- *Increasing precision.* This algorithm and the BFS-like approach represent two extremes. By moving towards the middle, we can increase precision, while sacrificing the running time. Of course, it is worthwhile doing so, as long as decrease in efficiency is reasonable compared to the savings we gain during later stages of verification.

Bench	No RA				MLP			
mark	bits	gates	SAT	BDD	bits	gates	SAT	BDD
PPP	1435	24628	TO(69)	TO(169)	445	18955	TO(89)	TO(216)
TCAS	1481	8792	2.1s	278.7s	765	5550	1.7s	109.6s
BKRY	198	1515	70.2s	16.1s	24	449	27.7s	0.6s
ARRY	359	5212	3.1s	47.1s	160	2440	2.2s	7.8s

Table 5.1: MLP range analysis benchmarks

5.4 Experiments

We have implemented our range analysis techniques in our prototype verification platform for C programs [33]. In this section, we report experimental results for the use of these techniques in verification of several benchmarks.

MLP-based and bounded range analysis. The experimental results are summarized in Table 5.1 and Table 5.2. For each benchmark, we include results for three methods: no range analysis; our MLP-based method described in Sections 5.1 and 5.2; and our bounded range analysis method (BoundedRA) described in Section 5.3. For each method, we include the model size (number of state bits and number of Boolean connectives), and the timing results for two verification methods: SAT-based bounded model checking and BDD-based unbounded model checking. The timing results show either time in seconds (numbers ending with `s`), or the maximum depth reached before one-hour timeout (TO(maxdepth)). For the bounded range analysis, the computed ranges are valid for traces of up to 2000 steps; the ranges are sound for all analyses described here, as none of the analyses exceed this depth.

In the PPP example we check correctness of an implementation of the point-to-point protocol with respect to its high-level specification. The TCAS example checks a safety property of an air traffic control software. BKRY checks mutual

Bench	No RA				BoundedRA			
mark	bits	gates	SAT	BDD	bits	gates	SAT	BDD
PPP	1435	24628	TO(69)	TO(169)	258	17909	TO(93)	TO(273)
TCAS	1481	8792	2.1s	278.7s	336	3912	1.7s	40.1s
BKRY	198	1515	70.2s	16.1s	24	449	27.7s	0.6s
ARRY	359	5212	3.1s	47.1s	42	1369	1.1s	3.3s

Table 5.2: Bounded range analysis benchmarks

exclusion for a faulty implementation of Lamport's bakery algorithm. ARRY is an array-manipulation example involving several nested loops, that checks for out-of-bounds array accesses. The results show that range analysis can significantly reduce the complexity of verification. The number of state bits, the size of the Boolean circuit representing the transition relation, and the verification time are all noticeably reduced. The results also show that bounded range analysis can give much better results than unbounded range analysis based on MLP solvers.

Range analysis for abstraction refinement. We have also tested the effect of range analysis on predicate abstraction, on examples from air traffic control software (TCAS) and a device driver (SERIAL). On examples from the TCAS benchmark, the total reduction in number of state bits was 61%. For four examples of predicate abstraction, on average we saved 49% on abstraction time, and 49.5% on refinement time in our current predicate localization framework [35]. For a previous predicate abstraction implementation based on SAT-based enumeration of the transition relation [44] we found similar performance improvements due to the fact that the concrete variable representation is significantly reduced with range analysis, thus resulting in faster runtime per solution. For 13 examples of predicate abstraction from the SERIAL benchmark, the use of range analysis reduced the total abstraction time by

35.7% and the total refinement time by 29.4%. The time for range analysis itself was almost always negligible; in one case though, the time spent on range analysis offset the savings in verification time.

Comparison with LP-based range analysis We have also compared our MLP-based method with the LP-based method of [55]. On one of our examples, the MLP-based method removed 74% of state bits compared to 37% with the LP-based method; the analysis run-time with BDDs was 26s with the MLP-based method compared to 265s with the LP-based method. The use of MLP did not affect range analysis runtime, which remained a negligible fraction of verification time.

5.5 Conclusions

In this chapter, we proposed the use of lightweight and efficient range analysis techniques for improving the performance of software verification. Our main method formulates the range analysis problem as a system of inequality constraints between symbolic bound polynomials. It then reduces the constraint to an MLP problem.

A second approach called bounded range analysis was introduced for bounded depth analyses. Both methods have been implemented in our prototype verification platform [33]. We also described a technique for improving the quality of range analysis in the presence of arrays, and described how range analysis can improve verification performance for two different representations of arrays. Finally, we presented promising experimental results on a variety of software verification benchmarks.

Chapter 6

Static Analysis for Invalid Pointer Accesses

Static analysis techniques have been successful in analyzing large programs for typical programming errors such as array buffer overflows, use of uninitialized variables, access of memory through invalid pointers, locking violations and others [6, 12, 15, 20, 23, 28, 34, 66]. Function summaries are often employed to provide an inter-procedural application of this technique [54]. Although the use of function summaries is often less precise than analysis of inlined source code, it allows scaling of the analysis to large pieces of code such as the Linux kernel [65].

In recent work, function summary-based techniques for program analysis using SAT-techniques have been shown successful to increase intra-procedural precision of the analysis [64]. The Saturn tool was able to find violations of an alternating lock-unlock checker [65], and improve precision for the detection of memory leaks [63]. In this chapter, we also employ a SAT-based function summary approach to analyze programs for invalid pointer accesses. We have implemented this technique in F-Soft [32], which is an analysis framework for C programs for user-defined properties and standard programming bugs.

Overview of the Approach. The first step in our analysis is to parse the given program and automatically annotate it with monitoring statements that monitor validity of pointers and flag invalid pointer accesses as reachability of unsafe error blocks. For the program in Fig. 6.1, we introduce three Boolean validity flags `rValid`, `pValid`, and `qValid` for the pointer variables `r, p`, and `q`, respectively. We introduce the statement `rValid = r?1:0` after the call to `malloc` and `rValid = 0` after the call to `free`. In addition, we have three possibly invalid accesses in the program: `free(r), *p,` and `*q` resulting in the creation of three potentially reachable error blocks B_k, B_j, B_i respectively. In each error block, we set a corresponding reachability predicate $\texttt{error}_k$, $\texttt{error}_j$, and $\texttt{error}_i$ to `true`. These reachability predicates are initialized to `false`. Following this monitor generation step, we may use many model size reduction and simplification steps such as program slicing, to reduce the models being analyzed by the following steps. We then process a program bottom up, starting with leaf functions, computing function summaries in terms of a set of function interface predicates. To compute function summaries in terms of predicates, we use symbolic simulation to encapsulate path-sensitive behavior of a single function using BDDs. As an example, we create the following BDD for the predicate $\texttt{error}_i$ in function `foo` in Fig. 6.1: $\texttt{q?(qValid?error}_i\texttt{:1):error}_i$. At the BDD level, we only track simple variable-to-variable assignments accurately, and replace other assignments with fresh variables. Each new variable is interpreted later when we build a bit-level accurate Boolean problem for the SAT-based function summary computation. We also use equivalence checks between these abstracted expressions to limit the number of required unknowns. In particular, when we process the function `top` depicted in Fig. 6.1, we can match the condition `(r)` explicitly present in the function and the condition in the monitoring statement `rValid = r?1:0`. The BDD for `rValid` thus simplifies to 1 before the function calls. We also use an efficient BDD variable ordering to capture the function structure, and thus avoid BDD size explosion as well as expensive variable reordering operations.

```
void top(){
  int *r;
  r = malloc(4);
  if(r){
     bar(r);
     free(r);
  }
}

void bar(int *p){
  ···
  *p = ··· ;
  foo(p);
  ···
}

void foo(int *q){
  ···
  if(q) *q = ··· ;
  ···
}
```

Figure 6.1: A motivating program fragment

A critical problem in using function summaries for detection of invalid pointer accesses is that a pointer that is changed inside a function may be aliased to some other pointer that is not accessed in the function. On the other hand, for accuracy, any changes to an aliased pointer in a function should be made observable as part of its summary. We introduce *context-insensitive pointer updates* to summarize the effect of function calls on aliased variables that are not directly changed inside the function. These statements are unsound in principle, but in practice they often capture the typical use of pointers in real software.

We then simplify the analysis by pre-computing relationships between invalid pointer accesses during an *error hierarchy analysis*. We look for scenarios of related invalid pointer accesses, i.e. where an invalid pointer access in one location implies another invalid pointer access in the remaining function body or in some other function. If the access through `p` in function `bar` is valid, then it must also be valid inside `foo` (see Fig. 6.1). Therefore, we only consider the access in `bar`. Many such cases arise in practice, since each pointer access generates a unique error block, although different pointer accesses may refer to the same pointer. In addition, validity of different pointer variables may also be highly correlated.

Performing summarization using all predefined predicates can become infeasible, and is in most cases unnecessary. We employ *inter-procedural predicate clustering* to cope with the problem. Each cluster contains output predicates involving the validity of a pointer, and its relationship to the predicates concerning the return variable of the function. To this cluster, we add all input predicates that syntactically share variables with the transfer functions of the output predicates. Similarly, we create a separate cluster for each error monitoring predicate. We also employ inter-procedural information by expanding these clusters using the required clustering information of the callees. We also use many static analysis techniques - property-based program slicing, constant folding, range analysis, control-flow graph simplifications and optimizations, and fast invariant computation techniques - on the given program. These are useful in reducing the size of the verification model used by the SAT solver for computing the function summaries, thereby improving its performance and scalability.

Outline. In the following section, we briefly review the modeling of software programs that is implemented in F-Soft, and is the basis of our analysis here. Next we describe our SAT-based function summary approach to find typical invalid pointer accesses in software. Section 4 then discusses how we simplify the analysis using relationships between potential invalid pointer accesses. In Section 6.4, we describe an implementation of these ideas in the F-Soft framework, and show the efficacy of this approach on Linux kernel modules and proprietary software, before we conclude this chapter with some final remarks.

6.1 Software Modeling in F-Soft

Let's briefly review software modeling in F-Soft, which is discussed in more detail in Chapter 2, Section 2.6. We begin with full-fledged C and apply a series of code transformations, each of which simplifies complex expressions into smaller but equivalent subsets of C. This preprocessing phase handles pointers, arrays, and bounded heap/stack related memory accesses. We use auxiliary variables that correspond to pointed-to locations for pointer variables [58]. While this introduces some additional variables into the model of the software, it allows us to simplify any indirect read through a pointer by having a single live pointed-to location [30, 58].

We also perform a points-to analysis to determine for each memory access the set of variables that may be accessed at that program location. We then replace the accesses through pointers with the appropriate conditional expressions. The simplified program consists of only scalar variables of types Boolean, enumerated, integer or real. Each program step is represented as a set of parallel assignments to these variables. Throughout the modeling and analysis, we use simplification techniques to reduce the model size. These steps include program slicing [37, 61], constant folding [51], and could also include techniques such as range analysis [55, 68], and other numerical domain analysis techniques using octagons [49].

6.1.1 Pointer Validity Checking

In order to analyze the input program for invalid pointer accesses, we automatically instrument the program by adding new Boolean monitors to the program. We consider a base pointer to be *valid*, if the base pointer points to a correctly allocated memory address either on the heap or on the stack. A pointer expression is considered valid, if the base pointer of the expression is valid according to the above definition. Note that this means that we do not consider arithmetic offsets in pointer expressions; that is, arithmetic offsets do not change the validity of a pointer in our analysis. A `null` base pointer is considered to be invalid. However, it is possible to construct an expression that evaluates to `null`, but would be considered valid using the above definition.

For each pointer variable, including pointer elements of arrays, we add monitoring flags. For example, a pointer variable called `ptr` may be monitored using the Boolean flag `ptrValid`. Pointer assignments update these validity flags by copying the corresponding flag for the right hand side of an assignment. The validity flags are also updated by calls to `malloc`, by setting the flag to `true` if `malloc` succeeds, and `false` otherwise. Calls to `free` set the validity of a pointer to `false`. We also model commonly used library functions in C, such as string manipulating routines. For every memory access, such as `*p`, `p[i]`, or calls to `free`, we introduce monitoring error blocks, that correspond to programming errors in the original source code should any such block ever be reached. Our analysis thus translates to checking whether any error block can be reached in the monitored program, and if so, which ones in particular. Note, that we do not check for calls to `free` with pointer variables with non-zero offsets.

6.1.2 Context-Insensitive Pointer Updates

We process program functions in reverse topological order starting from leaf functions, that is functions that do not call any other functions.[1] Whenever a function is called from another function, we substitute the call with its computed summary. A critical problem in using function summaries for detection of invalid pointer accesses is that a pointer parameter (or a global pointer) that is changed inside a function may be aliased to some other pointer variable that is not accessed in the function. On the other hand, for accuracy, any changes to an aliased pointer in a function should be made observable as part of its summary.

There are two ways of handling this situation in F-Soft. We can add so called *aliasing statements* into the corresponding basic block which track indirect accesses to aliased pointers, even if they are out-of-scope. These are used due to F-Soft s' modeling of pointed-to locations [30, 58]. We only add these aliased statements for pointers that may be aliased to the changed pointer.[2] As an example of such statements, consider the program fragment depicted in Fig. 6.2. The statement `free(q)` in function `foo` is replaced by the assignments $\texttt{q}_{foo}\texttt{Valid = 0}$ and $\texttt{p}_{bar}\texttt{Valid = (p}_{bar}\texttt{==q}_{foo}\texttt{)? 0:p}_{bar}\texttt{Valid}$, and similarly for $\texttt{q}_{bar}$ even though `p` and `q` (of `bar`) are out-of-scope. Most other statements that track values of pointed-to locations are often not important and are sliced away. However, in some cases such as programs with arrays of pointers, these statements may not be sliced away. The problem with this approach is that the number of clusters and pointer comparison predicates becomes prohibitively large.

[1]We treat mutually recursive functions as a single functional entity.

[2]It should be mentioned that we first perform a fast Steensgaard-style pointer alias analysis [60] to limit the potential aliasing relationships that need to be considered.

```
void bar(int *p, int *q){          void foo(int  *q){
   ...                                ...
   foo(p);                            free(q);
   ...                                ...
}                                  }
```

Figure 6.2: A program fragment

Another option is to introduce approximating statements to the pointer validity flags not in the callee, but in the appropriate caller. We call this context-insensitive pointer updates. An update to a pointer function parameter `p` only propagates to other in-scope aliased pointers. The updates to all other pointers are delayed until we process the corresponding caller. In other words, we only update pointers whose actual aliasing relationship with `p` can be established without considering the calling context. As an example, consider again Fig. 6.2. Since we pass pointer `p` to `foo`, we are only interested in analyzing whether `p` or any aliased pointer `q` is indirectly invalidated using a call to `free`; that is, we do not have to worry about a call to `malloc` that may make `p` or any aliased pointer valid after the function call. Therefore, after the call to `foo`, we introduce the following statements into our model for function `bar`:

$$\mathtt{p}_{bar}\mathtt{Valid} = \quad \mathtt{!q}_{foo}\mathtt{Valid}? \quad 0 : \mathtt{p}_{bar}\mathtt{Valid}, \qquad (6.1)$$

$$\mathtt{q}_{bar}\mathtt{Valid} = \quad (\mathtt{p}_{bar}\mathtt{\_old==q}_{bar}\mathtt{\_old\ \&\&\ !q}_{foo}\mathtt{Valid})? \quad 0 : \mathtt{q}_{bar}\mathtt{Valid}, \qquad (6.2)$$

where $\mathtt{p}_{bar}\mathtt{\_old}$ and $\mathtt{q}_{bar}\mathtt{\_old}$ are the values of `p` and `q` in `bar` before the call.

Statement (1) is meant to model the fact that the parameter `p` passed to `foo` may be freed inside `foo`. We approximate the effect of the function call to `foo` on `p`, by saying that `p` was probably freed by `foo`, if, at the end of the execution of `foo`, parameter `q` of `foo` is invalid. Otherwise, that is, if `q` of `foo` is valid at the end of function `foo`, we do not change the validity of `p` inside function `bar`. The reason is that the validity of `q` of `foo` may be due to the fact that it was allocated inside `foo`, but that allocation does not change the validity of what `p` originally pointed to. Similarly, it could have been aliased to some valid pointer, which again should not be reflected onto `p`. The above statement is incorrect, however, if `q` of `foo` was first freed inside `foo`, and then allocated locally. In this scenario, we produce a model where the function call to `foo` would not change the validity of `p`, although it should have been invalidated. Note that we rely on programming styles that discourage such coding practice.

A similar argument can be made to understand statement (2), which models the call to `foo` on the potentially aliased pointer `q`. Intuitively, we are saying that if `p` and `q` are aliased before the call, and `p` is freed by `foo` then we also update the validity flag for `q`. We handle memory allocation in a similar fashion, when modeling function calls where a pointer is passed by reference. Unlike the aliasing statements approach, these context-insensitive updates are unsound. In particular, aliasing relationship between pointers may change during function calls. A simple separate analysis is needed to detect such situations. However, this should not happen often since re-aliasing a parameter before performing `free` or `malloc` is a very undesirable programming practice. Context-insensitive pointer updates allow us to scale the analysis by guaranteeing that function summaries need only be in terms of function parameters and globals.

6.2 Function Summary Computation

A function summary is a relation between the input and output values of parameters and globals. We use symbolic simulation to capture the effect of a function, and develop an efficient BDD-based approach to represent loop-free programs. For scalability purposes, we compute summaries of functions in terms of predicates using SAT-based enumeration techniques [44].

6.2.1 Functional Symbolic Simulation

In this section, we describe how we use BDDs to symbolically simulate the behavior of a function. BDDs provide a good balance in addressing the following concerns: size of representation, canonicity of the representation to allow equivalence checks of expressions, and simplification of expressions. We have developed an algorithm that computes a good variable ordering, and we guarantee that all expressions are linear in the code size as long as we do not have any `goto`s. BDDs have been used to encode path-sensitive information efficiently in other contexts before, such as to perform symbolic RTL simulations [43]. There are other interesting alternatives for representing expressions such as multi-terminal binary decision diagrams (MTBDDs) [25] and Free Conditional Expression Diagrams (FCEDs) [26]. MTBDDs provide a high simplification degree, but can be exponential, while FCEDs do not provide canonicity.

For each basic block B_i, we compute a *guard* $\mathcal{G}_i$ which is an abstracted Boolean formula that represents the condition under which B_i is reachable from the function entry point. These guards are stored as BDDs. For each variable x and block B_i, we define a *transfer function* $\mathcal{F}_i^x$ which is a formula over the input variables of the function that represents the value of the variable x at the beginning of B_i. Similarly, we use $\widehat{\mathcal{F}}_i^x$ to denote the value of x at the end of B_i. The main reason to use BDDs is to efficiently represent path-sensitive information and perform high-level simplification. We employ two interconnected techniques to achieve that. First, at the BDD level we only track simple variable-to-variable assignments accurately, and replace all other assignments with fresh variables in a *blobbing* process [57]. These variables are interpreted later by building a bit-level accurate Boolean model for the SAT-solver. Second, we use these blobbed BDDs to find equivalent expressions to limit the number of required unknowns. We also introduce new BDD variables for program conditions.

The equivalence check can result in significant BDD simplification. Consider the example in Fig. 6.3 taken from [38]. The example shows that often there is a need to correlate sub-paths through a function, such as in the case of nested or sequential `if`-statements. In the example, there are two conditions which both refer to `p` and which can be found equivalent. Therefore, the transfer function for the `lock` variable at the `assert` location is found to be c_1, where c_1 is a symbolic variable representing the constant one. Consequently, the assertion `lock==1` is simplified to `true`. In addition, at the exit of the function `example2` the transfer function for the `lock` variable is c_0, where c_0 is a symbolic variable representing the constant zero.

In Fig. 6.4 and Fig. 6.5 we show how to compute guards and transfer functions for a basic block, assuming that all its parents have already been processed. We repeatedly apply the process for each block in topological order starting from the entry block. Note that at this stage function calls are handled by introducing fresh unknowns for variables potentially updated by the function call. Capturing the actual semantics of the function call is performed later by the SAT-solver.

void $example2()$	Transfer Function for `lock`
0. {	0. $lock_0$
1. $lock = 0;$	1. c_0
2. $if(p)\ lock = 1;$	2. $p_0?c_1 : c_0$
3. $\cdots$	3. $p_0?c_1 : c_0$
4. $if(p)\{$	4. c_1, in the `then`-branch; c_0, in the `else`-branch
5. $assert(lock == 1)$	5. c_1
6. $lock = 0;$	6. c_0
7. }	7. c_0
8. }	8. c_0

Figure 6.3: Using BDDs to slice and simplify a program.

- **Initialization**: All guards are initialized to true, $\mathcal{G}_0 = 1$
- **Flow Merging**: Let $Pre(B_i)$ denote the set of indexes of all the predecessors of the block B_i, and g_{ij} denote the guard for going from the block $B_j \in Pre(B_i)$ to B_i.

$$\mathcal{G}_i = \bigvee_{j \in Pre(B_i)} \mathcal{G}_j \wedge g_{ij}.$$

Figure 6.4: Computing Guards as Blobbed BDDs

Handling Loops. In the above discussion, we omitted how loops are handled. After F-Soft completes its preprocessing including program slicing and CFG simplifications, we currently unroll all remaining loops a fixed number of times. This approach seems adequate for invalid pointer access analysis. Nevertheless, we follow special rules when handling loops, which greatly increase soundness:

- A fresh unknown is introduced for each expression and each loop unrolling.
- We do not interpret unknowns introduced for dirty expressions (i.e., an expression, whose value might be affected by a loop).

- **Initialization**: For each variable x, the transfer function is initialized to a fresh symbolic variable, $\mathcal{F}_0^x = x_0$
- **Flow Merging**:

  $$\mathcal{F}_i^x = \bigvee_{j \in Pre(B_i)} \mathcal{G}_j \wedge g_{ij} \wedge \widehat{\mathcal{F}}_j^x.$$

  In addition, for efficiency, we restrict the BDD for $\mathcal{F}_i^x$ by the guard $\mathcal{G}_i$, to minimize the size. In particular, when there is only one predecessor B_j, the guard $\mathcal{G}_i = \mathcal{G}_j \wedge g_{ij}$, and after restricting the transfer function $\mathcal{F}_i^x = \mathcal{G}_j \wedge g_{ij} \wedge \widehat{\mathcal{F}}_j^x$, we obtain $\mathcal{F}_i^x = \widehat{\mathcal{F}}_j^x$.
- **Function Calls**: For each function call, represented by the summary edge (B_i, B_j), we *reset* each variable x that might be updated by the callee to a fresh symbolic variable u.

  $$\mathcal{F}_i^x = u.$$
- **Assignments**: Since all of the assignments are only of the form $x := y$, we just let

  $$\widehat{\mathcal{F}}_i^x = \mathcal{F}_i^y.$$

  If a variable is not changed then:

  $$\widehat{\mathcal{F}}_i^x = \mathcal{F}_i^x.$$

Figure 6.5: Computing Transfer Functions as Blobbed BDDs

- Dirty expressions are only matched within the same loop body.

This over-approximative handling of individual loop unwindings converges fast, and can easily be extended until a sound fixpoint has been reached for all loops. Another approach would be to handle loops as functions themselves, thus allowing the same procedure that is used to summarize functions to handle loops soundly. Nevertheless, we currently use an unsound unwinding approach that works well in practice to catch most bugs, while providing efficiency. While our intra-procedural algorithm is similar to the one used in Saturn [64], there are several important distinctions. We use BDDs to represent both the guards and the transfer functions, while Saturn uses BDDs to represent guards only. In addition, we do not model individual bits using transfer functions. Indeed, taking into account the blobbing, all the bits are updated uniformly to derive a "word-level" transfer function. We also use various preprocessing analysis steps, such as program slicing and constant folding, to reduce the model size.

Variable Ordering. The variable order is crucial for a succinct BDD representation. In our framework we can efficiently compute a good variable ordering by using the following two basic rules:

- The ordering among variables that correspond to conditions follows the control flow of the program itself. Whenever two branches of a conditional statement meet at a basic block, the BDD variable that represents the condition should have the lowest ordering (i.e., the symbolic variable representing the condition is the first one we split on) at the join basic block.
- We assign a lower ordering to variables representing conditions than to variables representing assignments. It is interesting to note that the ordering among variables that correspond to assignments is irrelevant, since assignments naturally correspond to leaf nodes in our BDD representation. This intuitively corresponds to so called reaching definitions of variables for which we build transfer functions.

Theorem 6 *For a loop-free, goto-free program the transition functions constructed by the algorithm described in Fig. 6.4 and Fig. 6.5, have the total number of BDD nodes $O(V * N)$, where V is the number of variables and N is the number of basic blocks.*

We give a proof by induction on the structure of the program. The base case when we just have one basic block or sequential composition of basic blocks is trivial. Indeed, all the BDDs will have just one node representing either the initial value of some variable or the unknown that is introduced for the right hand side of some of the assignments. The only other two cases that we need to consider are described below.

Sequential Composition. Assume that we compose two code fragments of sizes N_1 and N_2 respectively. Our algorithm first builds the transfer function to represent the first fragment and by applying the inductive assumption we find that the total number of nodes is $O(V * N_1)$. When the algorithm proceeds to the second section the transfer function of the first fragment are essentially used as initial values for the second fragment, and, again, applying inductive assumption, we add at most $O(V * N_2)$ new nodes.

Conditional Statements. The only remaining possibility is when two fragments of sizes N_1 and N_2 are composed using a conditional statement. For simplicity, we assume that we have unique branch and merge blocks for each such statement, so the number of blocks in the composed program is $N = N_1 + N_2 + 1 + 1$. Since the blocks are processed in topological order, before considering the merge block, we first build transfer functions for the two branches each resulting in $O(V*N_1)$ and $O(V*N_2)$ new nodes respectively. The only remaining step is the merging of the transfer functions according to the formula:

$$\mathcal{F}^x_{composed} = (\neg g \wedge \mathcal{F}^x_{left}) \bigvee (g \wedge \mathcal{F}^x_{right}), \text{ where } g \text{ represents the condition.}$$

Since g is guaranteed to have the lowest index in our BDD order, the merge can introduce at most one new node (the one that splits on g) for each resulting transfer function. The total number of nodes thus is $O(V * N_1 + V * N_2 + V) = O(V * N)$.

As we have shown the heuristics produce linear sized BDDs for any `goto`-free program and they also works well in practice for an arbitrary CFG, as evidenced by the experiments presented in Sec. 6.4. Also note that we use the variable order that is computed in the exit block of a function as the global order for the function.

6.2.2 Summary Computation

Having computed all blobbed transfer functions for the exit block, we first interpret the unknowns. These interpreted expressions represent a transition relation $R(V, V')$, which expresses the relation between function interface variables V (including globals) at the beginning of a function call and the values at the function return denoted as V'. Since such a transition relation is too detailed, we compute an over-approximation of R based on a predefined set of input and output predicates as the function summary.

Formally, a function summary is a relation $\mathcal{R}(P_{in}, P_{out})$, where P_{in} is a set of input predicates over input variables V, and P_{out} is a set of output predicates over output variables V'. We define the set of input and output predicates using the following heuristics. For each pointer `p` that is a function parameter or is a global variable, we introduce two predicates `p==0` and `pValid` into both predicate sets (after renaming for P_{out}). Also, we introduce predicates on the return values of functions, and add them to P_{out}. In particular, we check for nullness of a return variable, as well as for the validity of the return variable, if it is a pointer. Finally, for every potential invalid pointer access represented as an error block B_i, we introduce a reachability predicate expressed as $\texttt{error}_i$.

Inter-procedural predicate clustering. Using the above rules we often obtain too many predicates. Computing $\mathcal{R}$ most accurately, that is performing summarization using all predicates seems to be infeasible and in most cases unnecessary. We employ predicate clustering techniques [36] to cope with the problem, which essentially means that we compute decomposed summaries that over-approximate $\mathcal{R}$ and thus further over-approximate R. A separate predicate cluster is created for each pointer variable. Each cluster contains the output predicates involving the nullness of the pointer, its validity, and its relationship to the predicate(s) concerning the return variable of the function. Furthermore, we add all input predicates that syntactically share variables with the transfer functions of the aforementioned output predicates. Similarly, we create a cluster for each illegal pointer access error monitoring predicate, and all the input predicates that syntactically share variables with the monitor.

Note that the above does not capture inter-procedural variable dependencies since a variable changed by a callee is reset with an unknown. Therefore, we expand our clusters using the clustering information of the callees. We use the clustering information of the callee by interpreting the input and output predicates according to the context of the call, and adding input predicates into the clusters that are relevant according to the summaries of the callee.

At the last stage of summary computation we apply the summaries of the callees. For each cluster of a callee we create a Boolean formula relating the interpretations of the input and output predicates using accurate bit-level modeling. We then use SAT-based abstraction enumeration techniques [44] to compute the abstract transition relations for all clusters.

Eager and a lazy summary application. As one can observe the summaries of the callees are not "inlined" (i.e., applied eagerly) using possibly an exponential number of assignments. All potentially updated variables are simply reassigned with unknowns, whose values are restricted by the corresponding summaries. The restriction happens when we compute the summary for the caller and does not effect the computation of transfer functions. For a comparison between the eager and lazy approaches see Example 7.

Example 7

To illustrate the difference between and eager and a lazy summary application assume that we have two functions `bar` and `foo` as shown in Fig. 6.6. Assume that we have two predicates $P_0 := (x == 0)$ and $P_1 := (y == 0)$. Then the summary for `foo` can be expressed as $P_0' == P_0 \wedge P_1' == P_0$, where P_i' stands for the value of the predicate P_i after execution of the function.

```
void bar(···){
  bool x, y;
  x = 1;
  foo(x, y);
}
```

```
void foo(bool x, bool &y){
  ···
    y = x;
  ···
}
```

Figure 6.6: A program fragment for Example 7.

In Fig. 6.7, we apply both approaches for our example. In the lazy approach every *assume* statement is essentially a clause to a SAT solver. In the eager approach, we do not directly create any additional clauses, but essentially encode this information using transition functions. However, there is a chance to apply simplifications to the transition functions, before we perform the enumeration. In this example, we can potentially discover that y is simply equal to 1 after the call to `foo`.

```
void $bar(\cdots)\{$
    bool $x, y$;
    $x = 1$;
    Interpret input predicates:
        $assume(P_0 == (1 == 0))$;
        $assume(P_1 == (y_0 == 0))$;
    Interpret output predicates:
        $assume(P'_0 == (1 == 0))$;
        $assume(P'_1 == (U == 0))$;
    Reset possibly changed variables:
        $y = U$;
    Apply summary of $foo$
        $assume(P'_0 == P_0 \wedge P'_1 == P_0)$;
$\}$
```

(a) Lazy Approach

```
void $bar(\cdots)\{$
    bool $x, y$;
    $x = 1$;
    Try all combinations of the predicates
    when x and y are changed:
    $if(x == 1 \;\&\&\; y = 0) \quad (x, y) = (1, 1)$;
    $if(x == 0 \;\&\&\; y = 1) \quad (x, y) = (0, 0)$;
$\}$
```

(b) Eager Approach

Figure 6.7: Lazy vs Eager Summary Instantiation

Range analysis. We also improve the efficiency of the SAT-based summary computation by limiting the number of bits that are required for variables. The number of bits needed to encode a variable depends on its range. For example, an integer variable is generally represented using 32 bits, while a `char` variable requires 8 bits. One distinguishing feature of our work is that we may perform a pre-processing step called *range analysis* [55], which allows us to reduce the bitwidths of many variables. As was discussed in Chapter 5, this is often crucial to succeed in enumerating the predicated abstract transition relation.

6.3 Error Hierarchy Analysis

In this section, we describe techniques to simplify the analysis by finding relationships between invalid pointer accesses. We look for scenarios where an invalid pointer access in one location implies another invalid pointer access in the remaining function body or in some other function. Many such cases arise in practice, since each pointer access generates a unique error block, although different pointer accesses may refer to the same pointer. In addition, validity of different pointer variables may also be highly correlated. Furthermore, since we perform a bottom-up function-level analysis, we process a called function body before we analyze its calling context. Thus, the function summary includes scenarios that are actually executed, as well as scenarios that are never executed. Many of such invalid scenarios are correlated and can be discovered using our error hierarchy analysis. This hierarchy analysis both decreases the model size and thus improves efficiency of the analysis, as well as results in significantly fewer warnings that need to be analyzed by the user.

6.3.1 Intra-procedural Error Hierarchy Analysis

We first try to identify related error blocks within a function, which is started before the summary computation is performed. Assume B_i and B_j are two error blocks with the corresponding blobbed guards $\mathcal{G}_i$ and $\mathcal{G}_j$ represented as blobbed BDDs. We define a relation $\lesssim$ between error blocks, which corresponds to a logical implication between the BDDs representing the reachability of these basic blocks. If $\mathcal{G}_i \implies \mathcal{G}_j$, then we say $B_i \lesssim B_j$ and we do not consider B_i until we have resolved the reachability of B_j. In addition, even though the guards may contain blobbed expressions, the $\lesssim$-relation means that either B_j is reachable or B_i is unreachable. It should be noted though that due to using blobbed BBDs we may not discover all actual implications.

At the end of the analysis, if B_j is proved to correspond to a valid dereferencing, either automatically or with the help of the user, we can immediately guarantee the safety of B_i. Otherwise, if B_j is a confirmed bug we can present B_i for further investigation. Note that if $B_i \lesssim B_j$ and $B_j \lesssim B_i$, then we prefer to consider the basic block with a *shorter path* from the function entry point in the control flow graph of the function.

6.3.2 Inter-procedural Error Hierarchy Analysis

We can extend the idea from Section 6.3.1 to the inter-procedural case. Consider the example in Fig. 6.1. It is intuitively clear that if the dereferencing of pointer `p` in function `bar` is valid, then it must also be valid inside `foo`. To find such inter-procedural relationships, we rely on the predicated function summaries. Suppose B_i is the error block corresponding to the dereferencing of `q` in function `foo`, and B_j is the error block to the dereferencing of `p` in `bar`. The summary for `bar` includes individual clusters for the predicates $\texttt{error}_i$ and $\texttt{error}_j$. Denote the corresponding function summaries as T_i and T_j, respectively. Then, T_j can be expressed as $\neg\texttt{pValid} \implies \texttt{error}_j{}' \wedge \texttt{pValid} \implies \mathit{Pres}(\texttt{error}_j)$, where $\mathit{Pres}(p)$ stands for $p' \iff p$. T_i can be expressed as a conjunction of $(p \neq 0 \ \wedge \ \neg\texttt{pValid}) \implies \texttt{error}_i{}'$ and $(p == 0 \vee \texttt{pValid}) \implies \mathit{Pres}(\texttt{error}_i)$. From these summaries it follows that if we can show that the access at B_j is valid, then `pValid` must hold and consequently the access at B_i must also be valid.

More formally, suppose B_i and B_j are two error blocks with the corresponding transition relations T_i and T_j. If $(T_i \wedge T_j \wedge \neg\texttt{error}_j{}') \implies \mathit{Pres}(\texttt{error}_i)$, then we say that $B_i \lessapprox B_j$ and we do not consider T_i until we have resolved the reachability of B_j. If we are able to prove the access at B_j (in `bar`) correct, then the access at B_i when called through `bar` must also be correct.

6.4 Experimental Results

In this section, we describe our experimental evaluation for the analysis of invalid pointer accesses in Linux kernel modules (version 2.6.15) as well as on proprietary software. The examples encoded lnx1, lnx2, and lnx3 were chosen randomly from the Linux implementation of device drivers for sound cards. The examples, coded lnx4 and lnx5 were taken from a database of known Linux bugs. Finally, prop1 and prop2 are examples of proprietary software.

Benchmark	Model Description				Results		
	LOC	Blocks	Errors	Func.	Warn.	Bugs	Time(m)
lnx1 - ymf_ac97_init	7k	1775	86	26	10	0	1+9
lnx2 - ali_probe	19k	5546	234	54	15	0	20+70
lnx3 - via_init_one	13k	4995	134	40	10	0	20+20
lnx4 - snd_sbmixer_add_ctl	6k	1912	90	24	12	2	1+9
lnx5 - v9fs_create	8k	5602	124	20	6	1	5+9
prop1	13k	6207	495	12	13	0	3+70
prop2	3k	1037	35	10	7	1	1+2

Table 6.1: Experimental Results

Table 6.1 presents some experimental results on these benchmarks. We first provide the number of source code lines(`LOC`) used in each example, and then provide information on how many basic blocks(`Blocks`) are created in our CFG, how many of these are used to signal an invalid pointer access (`Errors`), and how many functions were processed in total (`Func`). Furthermore, we provide statistics on how many warnings were produced by our analysis, how many of these were actually invalid pointer accesses, and how much computation time was spend by our implementation. Our analysis declares 94% of potential pointer accesses correct. Most of the warnings reported by the analysis still represent false positives, which is largely due to sound modeling of potentially unbounded data structures such as linked lists. In addition, we did not provide the tool with any user input. On the other hand, while we do have a large percentage of false positives, we did not find any false proofs due to the unsoundness of the technique for any of the examples by inspection.

The run-time, shown in the last column contains two numbers, where the first represents the time it took F-Soft to construct the control flow graph and perform some preprocessing such as pointer aliasing analysis and program slicing. The second number is the time spend in our analysis for invalid pointer accesses starting with the construction of blobbed BDDs. Overall, the summary computation time is still quite high, but it can be significantly improved. This is due to the fact that F-Soft was designed as a highly precise model checking tool and not as front end for static analysis; in the future, we will develop relaxed software models targeted to increase the ratio of bugs per warning.

Table 6.2 gives more details about the performed experiments. In the first column we show the percentage of blocks that were sliced away by F-Soft preprocessing. The second column contains the percentage of error blocks that were proved unreachable for each of the examples using simple numeric domain invariant computations. While most of these error blocks could be proved unreachable by our method also, the above preprocessing steps reduces the model size. The example prep2_2, which is the same as example prep2_1 except that we omitted these simple preprocessing steps, highlights the difference, where the time spent in SAT-based function summary enumeration quadrupled. Note that for most other examples, we were not able to complete the analysis without these preprocessing steps using a three hour time limit.

Our next two columns show statistics on the BDD usage. In all experiments, even the largest transition function was linear in the size of the corresponding function. The largest transition function (taken from prop1), which has 466 BDD nodes, belongs to a function with more than 600 basic blocks. Moreover, using BDDs significantly helped our analysis by removing about 55% of the error blocks from consideration by analyzing intra-procedural error hierarchies. Example prop2_3 shows the case when we turned off the error hierarchy analysis for the example prop2_2. It should be noted that the analysis produced 8 additional warnings (not shown). Note, that we have not yet experimented with inter-procedural error hierarchy analysis.

Bench	F-Soft Analysis		BDD Statistics		SAT Statistics		
mark	Blocks Sliced	Err. Bl. Proved	Error Analysis $\lesssim$	MAX nodes	MAX Pred. per Cluster	MAX Pred. per Func.	SAT Time(m)
lnx1	74%	28%	45%	17	8	68	3.5
lnx2	78%	9%	41%	20	6	234	61.7
lnx3	61%	5%	42%	27	6	131	13.4
lnx4	13%	30%	65%	7	5	66	3.4
lnx5	89%	71%	25%	122	4	47	5.3
prop1	68%	36%	71%	466	6	329	55
prop2_1	67%	23%	37%	99	4	31	0.5
prop2_2	N/A	N/A	28%	101	5	48	2
prop2_3	N/A	N/A	N/A	101	5	48	2.3

Table 6.2: Detailed experimental results

In the last three columns we report statistics regarding SAT-based enumeration. We sometimes obtain hundreds of predicates to compute a function summary. In our experience, we require many predicates due to the underlying highly precise model build by the F-Soft front-end, such as for arrays of pointers, which means that we would not have been able to complete the SAT-based enumeration without predicate clustering. Fortunately, we were able to design simple but effective clustering heuristics for the analysis of invalid pointer accesses, which is evidenced by the fact that the largest cluster in all these benchmarks contains 8 predicates. Nevertheless, the SAT enumeration time, shown in the last column, remains the main bottleneck of our implementation for large benchmarks and requires further investigation.

6.5 Conclusions

In this chapter, we presented a SAT-based function summary analysis to detect invalid pointer accesses in programs. We propose many new techniques that are needed to address this problem in a fully automatic approach. We add context-insensitive pointer updates into caller functions to summarize the indirect effect of pointer manipulations inside a called function on aliased pointer variables. We use BDDs to represent a symbolic simulation of each function. We also present an inter-procedural predicate clustering technique based on variable dependencies in called function summaries. We perform an error hierarchy analysis to limit the number of warnings presented to the user, and also to increase efficiency of the analysis. We also employ other static analysis techniques such as program slicing to simplify and reduce the software model that needs to be considered. In the future, we wish to address the analysis of more complex properties, as well as improve the efficiency of various stages of the current implementation such as the SAT-based function summary computation.

Bibliography

[1] Accellera Organization, Inc. *Property Specification Language Reference Manual, Version 1.01*, 2003. http://www.accellera.org/.

[2] Franjo Ivancic Srihari Cadambi Malay Ganai Aarti Gupta Aleksandr Zaks, Ilya Shlyakhter and Pranav Ashar. Using range analysis for software verification. In *Proceedings of the 4th International Workshop on Software Verification and Validation (SVV 2006).* Computing Research Repository (CoRR), August 2006.

[3] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, Oct 1985.

[4] Aleksandr Zaks Amir Pnueli and Lenore Zuck. Monitoring interfaces for faults. In *Proceedings of the 5th Workshop on Runtime Verification (RV 2005)*, volume 144 of *Electronic Notes in Theoretical Computer Science*, pages 73–89, May 2006.

[5] E.M. Clarke andO. Grumberg and D.A. Peled. *Model checking.* MIT Press, 2000.

[6] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. *2002 IEEE Symposium on Security and Privacy*, 00:143, 2002.

[7] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In *Proc. VMCAI'04*, LNCS 2937, pages 44–57, 2004.

[8] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of the 36th ACM/IEEE Design Automation Conference*, pages 317–320, 1999.

[9] N. Bjorner, A. Browne, E. Chang, M. Colon, A. Kapur, Z. Manna, H. B. Sipma, and T. E. Uribe. STeP: deductive-algorithmic verification of reactive and real-time systems. In *Proc. CAV'96*, LNCS 1102, pages 415–418. Springer Verlag.

[10] J. R. Büchi. On a decision method in restricted second-order arithmetic. In *Proceedings Logic, Methodology and Philosophy of Sciences 1960*, Stanford, CA, 1962. Stanford University Press.

[11] A.K. Chandra, D.C. Kozen, and L.J. Stockmeyer. Alternation. *Journal of ACM*, 28(1):114–133, 1981.

[12] H. Chen and D. Wagner. MOPS: An infrastructure for examining security properties of software. In *9th ACM Conference on Computer and communications security (CCS'02)*, pages 235–244, New York, NY, USA, 2002. ACM Press.

[13] E.M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI-C programs using SAT. In *Model Checking for Dependable Software-Intensive Systems Workshop*, 2003.

[14] Bustan D., Fisman D., and Havlicek J. Automata Construction for PSL. 2005. http://www.wisdom.weizmann.ac.il/d̃ana/publicat/automta_constructionTR.pdf.

[15] E.A. Brewer D. Wagner, J.S. Foster and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Networking and Distributed System Security Symposium*, Feb 2000.

[16] Marcelo d'Amorim and Grigore Rosu. Efficient monitoring of omega-languages. In *CAV*, pages 364–378, 2005.

[17] Morton Davis. Infinite games of perfect information. In *Advances in game theory*, pages 85–101. Princeton Univ. Press, Princeton, N.J., 1964.

[18] L. de Alfaro, T. Henzinger, and R. Majumdar. From verification to control: dynamic programs for omega-regular objectives. In *Proc. LICS '01*, pages 279–290, 2001.

[19] L. de Alfaro and T.A. Henzinger. Concurrent omega-regular games. In *Proc. LICS'00*, pages 141–154. IEEE Computer Society Press, 2000.

[20] A. Deutsch. Static Verification Of Dynamic Properties. Technical report, 2004. `http://www.polyspace.com/white_papers.htm`.

[21] D. Drusinsky and M.T. Shing. Monitoring Temporal Logic Specifications Combined with Time Series Constraints. *JUCS*, 9(11):1261–1276, 2003.

[22] Cindy Eisner, Dana Fisman, John Havlicek, Michael Gordon, Anthony McIsaac, and David Van Campenhout. Formal Syntax and Semantics of PSL. 2003. http://www.wisdom.weizmann.ac.il/$\tilde{d}$ana/publicat/formal_semantics_standalone.pdf.

[23] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, 2002.

[24] Bernd Finkbeiner and Henny Sipma. Checking finite traces using alternating automata. In Klaus Havelund and Grigore Rosu, editors, *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier Science Publishers, 2001.

[25] M. Fujita and P.C. McGeer. Introduction to the special issue on multi-terminal binary decision diagrams. *Formal Methods in System Design*, 1997.

[26] S. Gulwani and G.C. Necula. Path-sensitive analysis for linear arithmetic and uninterpreted functions. In *11th Static Analysis Symposium*, volume 3148 of *LNCS*, pages 328–343. Springer-Verlag, August 2004.

[27] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, June 2001.

[28] G.J. Holzman. UNO: static source code checking for user-defined properties. In *World Conf. on Integrated Design and Process Technology (IDPT)*, June 2002.

[29] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, Reading, Massachussetts, 1979.

[30] F. Ivančić, I. Shlyakhter, A. Gupta, M.K. Ganai, V. Kahlon, C. Wang, and Z. Yang. Model checking C programs using F-Soft. In *Conference on Computer Design (ICCD)*. IEEE, October 2005.

[31] F. Ivančić, Z. Yang, M. Ganai, A. Gupta, and P. Ashar. Efficient SAT-based bounded model checking for software verification. In *Symposium on Leveraging Formal Methods in Applications*, 2004.

[32] F. Ivančić, Z. Yang, M.K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar. F-Soft: Software verification platform. In *Computer-Aided Verification (CAV)*, 2005.

[33] F. Ivančić, Z. Yang, I. Shlyakhter, M.K. Ganai, A. Gupta, and P. Ashar. F-Soft: Software verification platform. In *Computer-Aided Verification*, LNCS. Springer, 2005.

[34] T. Jaeger, A. Edwards, and X. Zhang. Consistency analysis of authorization hook placement in the linux security modules framework. *ACM Trans. Inf. Syst. Secur.*, 7(2):175–205, 2004.

[35] H. Jain, F. Ivančić, A. Gupta, and M.K. Ganai. Localization and register sharing for predicate abstraction. In *11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 3340 of *LNCS*, pages 397–412. Springer, 2005.

[36] H. Jain, D. Kroening, and E.M. Clarke. Verification of SpecC using predicate abstraction. In *MEMOCODE*, pages 7–16. IEEE, 2004.

[37] G. Jayaraman, V. P. Ranganath, and J. Hatcliff. Kaveri: Delivering the indus java program slicer to eclipse. In *Fundamental Approaches to Software Engineering (FASE)*, pages 269–272, 2005.

[38] R. Jhala, R. Majumdar, and R. Xu. Structural invariants. In *Symposium on Static Analysis (SAS)*, 2006.

[39] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In K. Etessami and S. K. Rajamani, editors, *17th Conference on Computer Aided Verification (CAV '05)*, pages 226–238. Springer-Verlag, 2005. LNCS 3576.

[40] Y. Kesten, A. Pnueli, and L. Raviv. Algorithmic verification of linear temporal logic specifications. In *Proc. 25th Int. Colloq. Aut. Lang. Prog.*, volume 1443 of *Lect. Notes in Comp. Sci.*, pages 1–16, 1998.

[41] Yonit Kesten, Nir Piterman, and Amir Pnueli. Bridging the Gap between Fair Simulation and Trace Inclusion. In *Proc. CAV'03*, LNCS 2725. Springer, 2003.

[42] Yonit Kesten and Amir Pnueli. A compositional approach to CTL* verification. *Theoretical Computer Science*, 331:397–428, 2005.

[43] A. Kölbl, J. Kukula, and R. Damiano. Symbolic RTL simulation. In *38th Conference on Design Automation (DAC)*, pages 47–52. ACM Press, 2001.

[44] S.K. Lahiri, R.E. Bryant, and B. Cook. A symbolic approach to predicate abstraction. In *Computer-Aided Verification (CAV)*, pages 141–153. Springer, 2003.

[45] D. Long, A. Browne, E. Clarke, S. Jha, and W. Marrero. An improved Algorithm for the Evaluation of Fixpoint expressions. In *Proc. CAV'94*, LNCS 818, pages 338–350. Springer, 1994.

[46] F.Y.C. Mang. *Games in Open Systems Verification and Synthesis.* PhD thesis, University of California, Berkeley, 2002.

[47] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.

[48] Donald A. Martin. Borel Determinacy. *Annals of Mathematics*, 102:363–371, 1975.

[49] A. Miné. The octagon abstract domain. In *AST in WCRE 2001*, pages 310 – 319. IEEE, October 2001.

[50] Satoru Miyano and Takeshi Hayashi. Alternating finite automata on ω-words. *Theoretical Computer Science*, 32:321–330, 1984.

[51] S.S. Muchnik. *Advanced Compiler Design and Implementation.* Addison Wesley, 1997.

[52] Amir Pnueli and Aleksandr Zaks. Psl model checking and run-time verification via testers. In *The 14th International Symposium on Formal Methods*, volume 4085 of *LNCS*, pages 573–586. Springer Berlin / Heidelberg, August 2006.

[53] Amir Pnueli, Aleksandr Zaks, and Lenore Zuck. Monitoring interfaces for faults. In H. Barringer, B. Finkbeiner, Y. Gurevich, and H. Sipma, editors, *Fifth International Workshop on Run-time Verification (RV)*, July 2005. Edinburgh, Scotland, UK.

[54] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Symposium on Principles of programming languages (POPL)*, pages 49–61. ACM Press, 1995.

[55] R. Rugina and M.C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Programming Language Design and Implementation (PLDI)*, pages 182–195, 2000.

[56] S. Safra. On the complexity of ω-automata. In *Proc. FOCS'88*, pages 319–327, White Plains, NY, 1988.

[57] J. Schwartz, E. Omodeo, and D. Cantone. *Computational Logics and Set Theory.* Springer, 2006.

[58] L. Séméria and G. de Micheli. SpC: Synthesis of pointers in C, application of pointer analysis to the behavioral synthesis from C. In *Conference on Computer-Aided Design (ICCD)*, pages 321–326. IEEE/ACM, November 1998.

[59] A.P. Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing*, 6:495–511, 1994.

[60] B. Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages (POPL)*, pages 32–41, 1996.

[61] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[62] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. First IEEE Symp. Logic in Comp. Sci.*, pages 332–344, 1986.

[63] Y. Xie and A. Aiken. Context- and path-sensitive memory leak detection. In *Foundation of Software Engineering (ESEC/SIGSOFT FSE)*, pages 115–125, 2005.

[64] Y. Xie and A. Aiken. Saturn: A SAT-based tool for bug detection. In *Computer-Aided Verification (CAV)*, pages 139–143, 2005.

[65] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *Principles of Programming Languages (POPL)*, pages 351–363, 2005.

[66] Y. Xie, A. Chou, and D. Engler. ARCHER: Using symbolic, path-sensitive analysis to detect memory access errors. In *Foundations of software engineering (FSE)*, pages 327–336, New York, NY, USA, 2003. ACM Press.

[67] C. Han Yang and David L. Dill. Validation with guided search of the state space. In *Proc. DAC'98*, June 1998. San Francisco, CA.

[68] A. Zaks, S. Cadambi, I. Shlyakhter, F. Ivančić, Z. Yang, M.K. Ganai, A. Gupta, and P. Ashar. Range analysis for software verification. In *Intern. Workshop on Software Verification and Validation (SVV)*, August 2006.

[69] Aleksandr Zaks, Franjo Ivancic, and Aarti Gupta. Static analysis for invalid pointer accesses using SAT-based techniques. *In preparation.*