INTERPOLATION METHODS FOR SYMBOLIC EXECUTION

CHU DUC HIEP

(BCompSc. Hons., 1st class)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

NUS GRADUATE SCHOOL FOR INTEGRATIVE SCIENCES AND ENGINEERING

NATIONAL UNIVERSITY OF SINGAPORE

2012

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

CHU DUC HIEP

8 April 2013

Acknowledgements

My deepest heartfelt gratitude goes to Tiffany for her unconditional love and support, for bearing with a busy and grumpy boyfriend, and now husband, for so many years.

Simply put, I'm blessed to have had Joxan Jaffar as my advisor. Throughout my Ph.D., he has been a constant source of inspiration and encouragement. It's really a privilege to always have him with me, to share not only the joy, the excitement, but also the frustration and disappointment regarding my research. Among many things I have learned and still learning from him, I deeply appreciate his values for clarity and simplicity, in which I now believe, that is how research and science should be done.

I have been fortunate enough to have several other mentors along the way. I am very grateful to Jin Song Dong and Siau Cheng Khoo, whose support and feedback have been particularly important. I am greatly indebted to Ben Leong, a very kind teaching supervisor, as well as a friend who has shared with me invaluable lessons in academia. I would like to thank Martin Sulzmann and Razvan Voicu for introducing me to research topics in programming languages since my undergraduate studies.

I would also like to thank Andrew Santosa, Jorge Navas, Vijay Murali, Thien Anh Dinh, Dinh Truong Huy Nguyen, Quang Loc Le, Minh Thai Trinh, and many more great friends and colleagues throughout the years, for contributing to a fun and exciting environment, in and out of office.

Last but not least, I would like to thank my parents and my elder brother, who have loved and inspired me all through my life. It was from them that I first developed my love for science and research.

Dedicated to my little Caroline

Contents

List of Tables I

List of Figures II

1 Introduction 1

1.1 Traditional Program Reasoning Techniques 2

1.2 Program Reasoning using Symbolic Execution 4

1.3 Thesis Contributions and Organization 7

2 Symbolic Execution with Interpolation 12

2.1 Symbolic Execution 14

2.2 Interpolation 17

I Program Path Analysis 23

3 Loop Unrolling 24

3.1 Contributions and Related Work 28

3.2 Path Analysis vs. Timing Model 31

3.3 Overview 33

3.4 Preliminaries 34

3.5 Motivating Examples 38

3.6 Symbolic Simulation Algorithm 44

3.7 Implementation Details 49
3.8 Experimental Evaluation 53
3.9 Other Related Work 57
3.10 Summary 58

4 Assertions 60
4.1 Related Work 68
4.2 Motivating Examples 72
4.3 Preliminaries 76
4.4 The Algorithm: Overview of the Two Phases 78
4.5 The Algorithm: Technical Description 84
4.6 Experimental Evaluation 92
4.7 Summary 95

II Safety Verification of Concurrent Programs 96

5 Combining State Interpolation and Partial Order Reduction 97
5.1 Related Work 99
5.2 Background and Discussions 102
5.3 State Interpolation Revisited 107
5.4 Property Driven POR 109
5.5 Synergy of SI and PDPOR 114
5.6 Implementation of PDPOR 121
5.7 Experiments 125
5.8 Summary 129

6 Complete Symmetry Reduction 130
6.1 Related Work 135
6.2 Preliminaries 137
6.3 Motivating Examples 141

6.4 Complete Symmetry Reduction Algorithm 144
6.5 Experimental Evaluation 150
6.6 Summary 152

7 Conclusion **154**
7.1 Summary 154
7.2 Concluding Remarks and Future Research 156

Bibliography **158**

SUMMARY

Symbolic execution is a method for program reasoning that uses symbolic values as inputs instead of actual data, and it represents the values of program variables as symbolic expressions of the input symbolic values. Symbolic execution was first developed for *program testing*, but it has been subsequently used for *program analysis* and *verification condition generation*, among others.

This thesis applies symbolic execution to two important and extremely hard application areas, namely *program path analysis* and *safety verification of concurrent programs*. The foremost challenge for symbolic execution is the exponential number of symbolic paths. This challenge is further aggravated due to the existence of loops (in program path analysis) and interleavings (in safety verification of concurrent programs). We address the challenge by building custom interpolation methods, of which the contributions can be summarized as follows:

- In program path analysis, our interpolation method allows us to summarize loop iterations and combine these *summarizations* in such a way that the cost of loop unrolling can just be *superlinear*. Informally, this means that the size of our symbolic execution tree is linear, even for nested loop programs of polynomial complexity. This is indeed a breakthrough in loop unrolling. We next propose a framework for program path analysis, which accommodates both path-sensitivity and *user assertions*. This has not been achieved before. The main challenge is that, a greedy treatment for loop in symbolic execution, while being fully compliant with assertions, can produce *unsound* results. We address this challenge by presenting a novel two-phase algorithm, where in each phase, we separately deal with *infeasible* paths and paths *blocked* by assertions.

- In safety verification of concurrent programs, simple *state interpolation* (e.g., in SMT or CEGAR) is no longer applicable. This is due to the astronomically

large state space resulted from process interleavings. In this domain, however, the most established techniques for state space reduction are *partial order reduction* (POR) and *symmetry reduction*. We contribute by weakening these traditional concepts, using the concept of interpolation, so that reduction now can be *property dependent*. Specifically, we first generalize traditional POR to property driven partial order reduction (PDPOR), by replacing the concept of *trace equivalence* with the concept of *trace coverage*. We then introduce a framework which *synergistically* combines the power of both state interpolation and PDPOR. Consequently, we achieve significantly better reduction than the state-of-the-art. We also introduce the notion of *weak symmetry* which allows for more symmetry than the notions used in the literature. Weak symmetry is defined relatively to the target safety property. The key idea is to perform *symmetric transformations* of state interpolants, on demand, and use them for pruning. Our method, when employed with an interpolation algorithm which is monotonic, can exploit weak symmetry *completely*. As a result, our work also breaks new ground in the realm of symmetry reduction.

List of Tables

3.1 WCET Benchmark Programs 53

3.2 Experiments on WCET Benchmark Programs 55

4.1 Experiments with and without Assertions 93

5.1 Experiments on Producers/Consumer Example 126

5.2 Experiments on Sum-of-ids Example 126

5.3 Experiments on Dining Philosophers and Bakery Algorithm 127

5.4 Experiments on Programs from ICSE11 128

6.1 Experiments on Dining Philosophers 150

6.2 Experiments on Reader-Writer Protocol 151

6.3 Experiments on Sum-of-ids Example 152

6.4 Experiments on Bakery Algorithm 152

List of Figures

1.1 A Simple Loop with Exponential Number of Paths 8

2.1 Transition System and Its Graph Representation 13
2.2 Performing Symbolic Execution 16
2.3 Interpolation and Witness for Analysis 20

3.1 Challenging Program Patterns 26
3.2 Iteration Abstraction and Summarizations of Loop 34
3.3 From a C Program to its Transition System 35
3.4 Infeasible Paths in Analyses 38
3.5 Witnesses Improve Precision 40
3.6 Superlinear Analysis of bubblesort 42
3.7 Symbolic Simulation Algorithm: Main Function 44
3.8 Symbolic Simulation Algorithm: Helper Functions 45

4.1 Need for Path Sensitivity 61
4.2 Assertions are Essential 63
4.3 Complying with Assertions in Loop Unrolling is Hard 65
4.4 Assertions in IPET 72
4.5 Assertions Alone Are Not Enough 73
4.6 Assertions Are Essential 74
4.7 Local Assertions 75

4.8 Memory High Watermark Analysis 76
4.9 Complying with Assertions in Loop Unrolling 79
4.10 Reduced Search Space in Phase 2 84
4.11 Two-phase Symbolic Simulation Algorithm 85
4.12 TransStep for Non-Cumulative Resource 91

5.1 Application of SI on 2 Closely Coupled Processes 107
5.2 State Pruning 108
5.3 Branch Pruning 110
5.4 New Persistent-Set Selective Search (DFS) 112
5.5 Algorithm Schema: A Framework for SI and POR (DFS) 115
5.6 Inductive Correctness 116
5.7 Two Producers and One Consumer 117
5.8 The Full Execution Tree 118
5.9 The Search Tree using Static Synergy Algorithm 119
5.10 Example on performance of PDPOR 123

6.1 Modified 3-process Reader-Writer Protocol 132
6.2 Sum-of-ids Example 134
6.3 Example: Awaits then Increments 138
6.4 Example: Assign `id` to `x[id]` 141
6.5 Example: Only Process #1 Increments 143
6.6 Complete Symmetry Reduction Algorithm (DFS) 145

Chapter 1

Introduction

> There are also two kinds of truths: truths of reasoning and truths of fact. Truths of reasoning are necessary and their opposite is impossible; those of fact are contingent and their opposite is possible.
>
> Gottfried Leibniz

"Software is hard", wrote Donald Knuth, author of the programming field's most respected textbooks [Knuth, 1997]. But why?

Indeed, this contradicts expectations, for building software requires neither large factories nor scarce resource. There have been many attempted explanations for this (e.g., [Brooks Jr., 1995]), however, fundamentally it is because: (1) software is designed, not built like a house; and (2) programming is a craft, not a science.

Given that "crafting software" is a human activity, errors occur. The situation is aggravated by the fact that large software system often has many levels of abstraction and no single programmer can possibly know all the details about the system. Consequently, it is extremely hard to control the correctness and performance of the overall software system. It has been now commonly accepted that software er-

rors are often too difficult to even detect, let alone isolate, identify, and correct. The most diligent and faithful applications of *random testing* can only mitigate this problem to a certain level. The core problem remains, as expressed by Dijkstra: "Program testing can be used to show the presence of bugs, but never to show their absence!" [Dijkstra, 1972].

Nowadays, every major software system that is released or sold, is almost guaranteed to contain bugs. On the other hand, having bugs in software is costly [NIST, 2002], and software failures have caused loss of lives in safety critical systems [Garfinkel, 2005]. As software has now become ubiquitous, the quest for reliable software has become increasingly important. Since the complexity of software system continues to escalate, so does the need for a rigorous methodology to *reason* about software system.

Program reasoning approaches use the means of mathematical and formal proof in order to discover and guarantee properties of programs. Reasoning is concerned with analyzing a program down to the smallest element, and then synthesizing an understanding of the entire program. As opposed to testing, reasoning can trace every path through a system, and consider every possible combination of circumstances, and be certain that nothing has been left out. This is possible because the method relies on mathematical proofs to assure the completeness and correctness of every step. What is actually achieved by reasoning is a mathematical proof that the program being studied satisfies its specification. If the specification is complete and correct, then the program is *guaranteed* to perform correctly.

1.1 Traditional Program Reasoning Techniques

Proving and discovering properties of programs have been well investigated. Here we mention program verification, model checking, and program analysis using abstract interpretation.

The seminal work of Floyd and Hoare [Floyd, 1967; Hoare, 1969] has pioneered

the area of program reasoning. In these early work, a calculus for proving program partial correctness was presented. This approach had the advantage of being compositional, in an assume-guarantee fashion. The calculus has later been extended to support total correctness reasoning, i.e., termination is also considered. Though Hoare calculus has been serving as the basis for propagation-based reasoning algorithms, which would operate either in a forward manner (strongest postcondition propagation), or in a backward manner (weakest precondition propagation), its limitation lies in the fact that it requires user-provided assertions and invariants, which in turn makes automation difficult to achieve.

Model checking [Clarke *et al.*, 1999] has experienced tremendous success with hardware verification, and verification of finite state systems, in recent years. The most important advantage of model checking is that it can be made completely automatic. Typically, the user only need to provide a high level representation of the model and the specification to be checked. The model checking algorithm will either terminate with the answer true, indicating that the model satisfies the specification, or give a counter-example execution in which the specification is not satisfied. The counter-examples are particularly important in diagnosing (and then fixing) subtle errors in complex transition systems. Model checking algorithm is also fast in general and can check partial specifications. However, when it comes to reasoning about software systems, which concerns (at least theoretically) infinite state systems, the restriction to a finite state space becomes a major disadvantage. In such case, abstraction techniques must be employed to produce a finite state approximation of the system. This approximation might result in the introduction and detection of spurious errors, i.e., false positives. To deal with this, recent techniques equipped with mechanisms for automatic abstraction refinement on-the-fly, usually referred to as the CEGAR family [Clarke *et al.*, 2000; Ball *et al.*, 2001], have been developed to to help distinguish between spurious and real errors.

Another major program reasoning approach is the abstract interpretation frame-

work [Cousot and Cousot, 1977]. This framework is frequently used inside compilers, to analyze programs in order to decide whether certain optimizations or transformations are applicable. Abstract interpretation simulates the execution of the program using an abstract domain that is Galois connected with the concrete semantic domain. In this process, one has to come up with a fixed abstract domain of finite lattice structure so that a set of concrete states of the program can now be approximated by an abstract state. This then results in a finite number of classes of program states. State space search is then performed on the finite classes. Abstract interpretation can be engineered to obtain efficient state-space traversal. Since the abstract domain is designed statically, however, the obtained level of accuracy could be *arbitrarily* low.

1.2 Program Reasoning using Symbolic Execution

We propose to develop a methodology in program reasoning founded on symbolic execution [King, 1976; Clarke, 1976]. Symbolic execution is a process which depicts different execution states of a program wherein each basic execution step can be described by a formula capturing the functional behavior of each basic operation, as opposed to a direct execution of the program (with fixed inputs). This process is intuitive because it resembles closely the human reasoning behind each execution step in question. The main advantage of symbolic execution is that it enables us to potentially obtain fully accurate reasoning because the propagation process is done in the exact symbolic domain.

Symbolic execution uses symbolic values as inputs instead of actual data, and it represents the values of program variables as symbolic expressions of the input symbolic values. A symbolic execution tree depicts all executed paths during the symbolic execution. A path condition is maintained for each path and it is a formula over the symbolic inputs built by accumulating constraints which those inputs must satisfy in order for execution to follow that path. A path is *infeasible* if its

path condition is unsatisfiable. Otherwise, the path is *feasible*. Symbolic execution was first developed for program testing [King, 1976], but it has been subsequently used for bug finding [Cadar *et al.*, 2006] and verification condition (VC) generation [Beckert *et al.*, 2007; Jacobs and Piessens, 2008], among others [Cadar *et al.*, 2011; Saswat, 2012].

Symbolic execution reasons about a program path-by-path. This may be superior to reasoning about a program, like dynamic testing does, input-by-input. However, to be practical, we first have to overcome the most fundamental challenge for symbolic execution, namely the *exponential number of symbolic paths*. The key concept to counter the path explosion problem is *interpolation* [Craig, 1955; McMillan, 2003]. We now briefly mention current state-of-the-arts in this direction.

Program Verification

The seminal work [Jaffar *et al.*, 2009] presented the method of *Abstraction Learning* (AL) for loop-free program fragments. This was contrasted as a *dual* to the current standard method of *CounterExample-Guided Abstraction Refinement* (CEGAR) [Clarke *et al.*, 2000; Ball *et al.*, 2001]. CEGAR starts with an abstract model of the program and if, in the ensuing abstract interpretation, an error is found, then a check of the error path is performed to determine if the path is indeed a real path (because abstraction admits "spurious" paths in general). If so, we have found an error; if not, then an examination of this path will be done in order to *refine* the abstraction, and then the whole process can be redone using the new abstraction. In AL, however, the technique starts with the concrete model of the program. Then, the model is checked for the desired property (verification phase) via symbolic execution. If a counterexample is found, then it must be a real error and hence, the program is unsafe. Otherwise, the program is safe.

The key idea in AL is to learn: it does this by eliminating from the concrete model those facts which are *irrelevant* or *too-specific* for proving the unreachability of the

error nodes. This learning phase consists of computing *interpolants* in the same spirit of *no-good learning* in SAT solvers. Informally, an interpolant is a generalization of a set of states for splitting between "good" and "bad" states.

Jaffar et al. [Jaffar *et al.*, 2011] then further enhance symbolic execution for handling unbounded loops but yet without losing the intrinsic benefits of symbolic execution. The method is based on three design principles: (1) abstract loops in order for symbolic execution to attempt to terminate, (2) preserve as much as possible the inherent benefits of symbolic execution (mainly, earlier detection of infeasible paths) by propagating the strongest loop invariants, whenever possible, and (3) refine progressively imprecise abstractions in order to avoid reporting false alarms.

Here we emphasize that the use of symbolic execution with interpolants for verification is thus similar to CEGAR [Henzinger *et al.*, 2004; McMillan, 2006], but symbolic execution has some benefits (see [McMillan, 2010]):

1. It does not explore *infeasible paths*, thus avoids the expensive refinement in CEGAR.

2. It avoids expensive *predicate image* computations of, for instance, the *Cartesian* [Ball *et al.*, 2004; Beyer *et al.*, 2007] and *Boolean* [Beyer *et al.*, 2009] abstract domains.

3. It can recover from *too-specific* abstractions in opposition to monotonic refinement schemes often used in CEGAR.

Program Analysis

A pioneering work using interpolation for analysis is [Jaffar *et al.*, 2008] on the specific problem of discovering the longest path in a loop-free problem, an instance of the NP-complete problem of Resource Constrained Shortest Path (RCSP). The novelty here was the use of *interpolants* [Jaffar *et al.*, 2009] and *witnesses*. When a path is analyzed, we extract an interpolant from the formula associated with the

symbolic states of its nodes. This is a more general formula stored at each node that preserves the relevant information in the path. When a subtree of paths is analyzed, we compute a *witness*, a formula which describes the (sub-)analysis of the tree. If one of the nodes is encountered in another path such that its current formula entails the previously computed interpolant and witness, we can avoid exploring the paths from that node. We call this step the *subsumption test*. Whenever the subsumption test fails (i.e., the entailment does not hold), symbolic execution will naturally perform node splitting and duplicate all successors of the node until the next merge point. Alternatively, if the test passes, a node merging is performed. The key insight is that the subsumed node *shares* the analysis results of the subsuming node, thus giving rise to the all-important computational optimization of *re-use*.

1.3 Thesis Contributions and Organization

This thesis applies symbolic execution to two important and extremely hard application areas of program reasoning, namely *program path analysis* and *safety verification of concurrent programs*. These two problem domains share a common characteristic that they require *reachability analysis* on the symbolic execution tree.

The thesis makes several contributions in the two areas. First, it gives a symbolic simulation framework which not only breaks new ground among loop unrolling techniques (Chapter 3, previously presented in [Chu and Jaffar, 2011]) but also is the first unrolling technique incorporating the use of user assertions (Chapter 4, previously presented in [Chu and Jaffar, 2012b]). Second, it extends the traditional concepts for state space reduction, namely partial order reduction and symmetry reduction, with the concept of interpolation so that pruning now can be *property dependent* (Chapter 5 and Chapter 6, previously presented in [Chu and Jaffar,] and [Chu and Jaffar, 2012a] respectively). Background material that our work builds upon is covered in Chapter 2.

Program Path Analysis

Symbolic execution with interpolation has been shown to be effective for the loop-free program fragments [Jaffar *et al.*, 2008]. However, the fundamental challenge of symbolic execution is much further aggravated due to the existence of loops. Let us quantify this matter with a concrete example.

```
for (i = 0; i < 100; i++) {
    if (rand() > 0.5)
        j++;
    else
        k++;
}
```

Figure 1.1: A Simple Loop with Exponential Number of Paths

Consider Fig. 1.1. In each of the 100 iterations, depending on the return value of the random function rand(), either j or k will be incremented. There are two possible outcomes during each loop iteration. Thus, the number of feasible program paths is 2^{100}. The first key observation is that, the number of feasible program paths is exponentially large. Second, because we are in fact performing symbolic execution, "the analysis time is always at least proportional to the actual execution of the input program. It leads to very long analysis time since symbolic execution is typically orders of magnitudes slower than native execution" [Wilhelm *et al.*, 2008].

In short, there are two fundamental issues caused by loops, which prevent symbolic execution from getting *exact* analysis: one involves the breadth; the other involves the depth of the symbolic execution tree.

In Chapter 3, we present our symbolic execution technique, applied to the problem of Worst Case Execution Time (WCET) *path analysis*. We address the first issue, namely breadth-wise, not only by using the concept of interpolation [Jaffar *et al.*, 2008], but also by applying *path merging* at the end of each loop iteration. The second issue, namely depth-wise, is resolved by *vertically combined* summarization.

A notable achievement is that the complexity of our analysis is often observed as *superlinear*, even for those loops which are classified as *complicated* loops. Informally, this means that the size of our symbolic execution tree for a nested loop program of polynomial complexity can just be linear. Therefore, symbolic execution can in fact be *asymptotically shorter* than a concrete execution. This is important because the cost of symbolic simulation is, clearly, far higher than concrete simulation.

Our work has broken new ground in loop unrolling techniques for program analysis. In term of accuracy, we achieve *exact* bounds for most of the benchmarks commonly used for evaluating WCET analysis. Importantly, our method *guarantees exact bound* in case of loop free programs, single-path programs (might contain loops), and programs where all path merges performed are not "destructive" [Thakur and Govindarajan, 2008a]. In term of scalability, our work overcomes the fundamental shortcomings of symbolic execution in regard of loop handling and works well with programs of small and medium size (up to 2K lines of code).

In Chapter 4, we propose a path analysis framework for general *resource usage*. Our framework supports not only analysis of *cumulative* resource but also analysis of *non-cumulative* resource such as memory high watermark. Most importantly, our framework is the first which accommodates both *path-sensitivity* and *user assertions* at the same time. We achieve this using a novel two-phase algorithm. In the first phase, we make use of our unrolling technique presented in Chapter 3 so that context propagation can be done precisely and efficiently. Our second phase tackles the *combinatorial* explosion, due to the requirement of being fully path-sensitive wrt. the provided *user assertions*, by employing an adaptation of dynamic programming with interpolants [Jaffar *et al.*, 2008]. The novelty lies in the significant simplification of program paths achieved at the end of phase 1, which makes [Jaffar *et al.*, 2008] now become applicable.

Safety Verification of Concurrent Programs

Verification of concurrent programs is extremely hard due to the state space explosion caused by interleavings of transitions from different processes.

Symbolic execution with interpolation, also referred to as state interpolation, (SI) has been shown to be effective for verification of sequential programs. In SI [Jaffar *et al.*, 2009; Jaffar *et al.*, 2011], a node at program point ℓ in the reachability tree can be pruned, if its context is subsumed by the interpolant computed earlier for the same program point ℓ. Therefore, even in the best case scenario, the number of states explored by a SI method must still be at least the number of all *distinct* program points. Since the number of global program points is the product of the numbers of program points in each process, in the setting of concurrent programs, exploring each distinct global program point once might already be considered *prohibitive*. In short, symbolic execution with interpolation (SI) *alone* is not efficient enough for verification of concurrent programs.

In the literature, two established concepts to reduce interleavings in verification of concurrent programs are *partial order reduction* (POR) and *symmetry reduction*. POR exploits the equivalence of interleavings of 'independent' transitions, i.e., two transitions are independent if their consecutive occurrences in a trace can be swapped without changing the final state. In other words, POR-related methods prune away *redundant* process interleavings in a sense that, for each Mazurkiewicz [Mazurkiewicz, 1986] trace equivalence class of interleavings, if a representative has been checked, the remaining ones are regarded as redundant. Symmetry reduction, on the other hand, exploits the similarity between processes in the concurrent system. In the global state space, this similarity gives rise to classes of states, each contains states which are *transformable* into one another via some permutation. The intuition for reduction is that we should check only one representative state for each of such class. Note, however, that both traditional POR and symmetry reduction are of little (if at all) *sensitive* wrt. the target safety property.

In Chapter 5, we first contribute by further weakening the concept of Partial Order Reduction to *Property Driven Partial Order Reduction* (PDPOR) — which is now *property dependent* — in order to adapt it for a symbolic execution framework with abstraction. This is made possible by introducing the concept of *trace coverage*, a generalization of the traditional concept of Mazurkiewicz trace equivalence. The main contribution of this Chapter, however, is a framework that synergistically combines state interpolation and PDPOR so that the sum is more than its parts.

Finally, in Chapter 6, we enhance the concept of symmetry reduction. Traditional symmetry reduction techniques rely on an idealistic assumption that processes are *indistinguishable*. Because this assumption excludes many realistic systems, there is a recent trend to consider systems of non-identical processes, but where the processes are *sufficiently similar* that the original gains of symmetry reduction can still be obtained, even though this necessitates an intricate step of detecting symmetry in the state exploration.

Here we present a general method for its application, restricted to verification of safety properties, but *without* any prior knowledge about global symmetry. We start by using a notion of *weak symmetry* which allows for more reduction than in previous notions of symmetry. This notion is relative to the target safety property. The key idea is to perform symmetric transformations on *state interpolants*, on demand, and use them for pruning. Our method naturally favors "quite symmetric" systems: more similarity among the processes leads to greater pruning of the tree. The main result is that the method is *complete* wrt. weak symmetry: it only considers states which are not weakly symmetric to an already encountered state.

Chapter 2

Symbolic Execution with Interpolation

> If history repeats itself, and the unexpected always happens, how incapable must Man be of learning from experience.
>
> George Bernard Shaw

We restrict our presentation to a simple imperative programming language, where all basic operations are either void operations, assignments, or assume operations. The set of all program variables is denoted by *Vars*. A *void* operation takes the usual semantic: it only changes the program location. An *assignment* x := e corresponds to assign the evaluation of the expression e to the variable x. In the *assume* operation, assume(c), if the conditional expression c evaluates to *true*, then the program continues, otherwise it halts. The set of operations is denoted by *Ops*.

We model a program by a *transition system*. A transition system $\mathcal{P}$ is a triple $\langle \mathcal{L}, l_0, \longrightarrow \rangle$ where $\mathcal{L}$ is the set of program points and $l_0 \in \mathcal{L}$ is the *unique* initial program point. $\longrightarrow \subseteq \mathcal{L} \times \mathcal{L} \times Ops$ is the transition relation that relates a state to its (possible) successors by executing the operations. This transition relation

models the operations that are executed when control flows from one program point to another. We shall use $\ell \xrightarrow{\mathsf{op}} \ell'$ to denote a transition relation from $\ell \in \mathcal{L}$ to $\ell' \in \mathcal{L}$ executing the operation $\mathsf{op} \in Ops$.

A transition system naturally constitutes a directed graph, where each node represents a program point and edges are defined by the relation $\longrightarrow$. This graph is similar to (but not the same as) the control flow graph of a program.

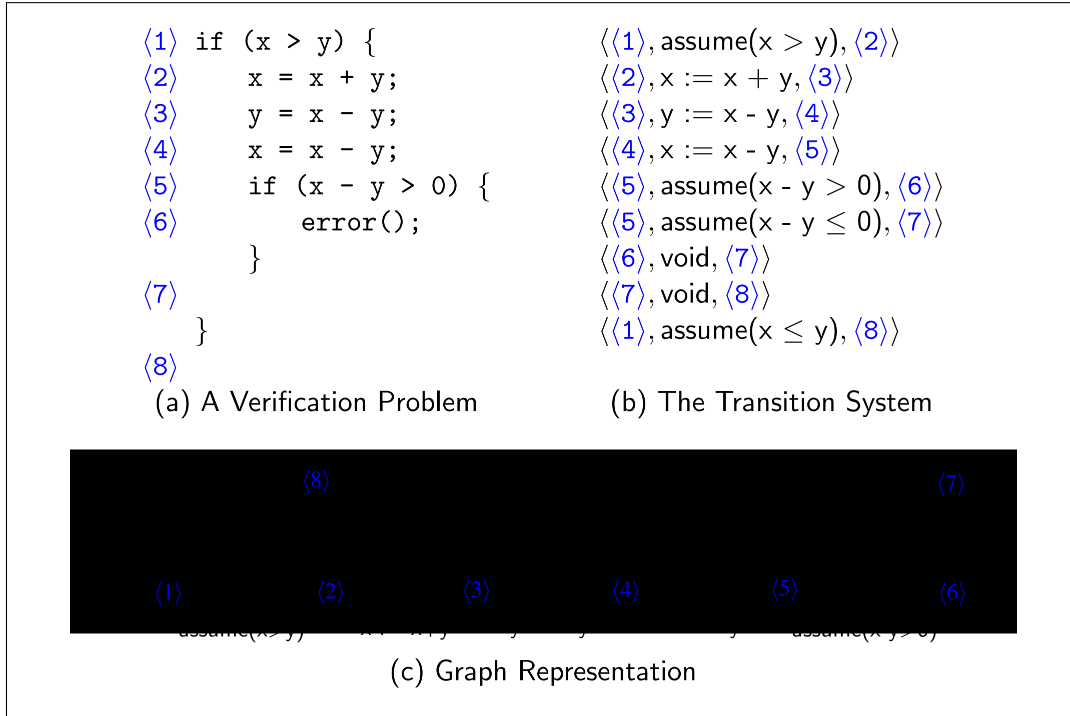


Figure 2.1: Transition System and Its Graph Representation

EXAMPLE 2.1 *(Transition System)*: Consider the verification problem in Fig. 2.1(a) where we want to prove that program point ⟨6⟩ is *unreachable*. This program fragment is taken from [Saswat, 2012]. The translated transition system is as in Fig. 2.1(b) and the corresponding directed graph is in Fig. 2.1(c). Note that the displays of void operations are unnecessary and we will omit them from now on.

2.1 Symbolic Execution

One advantage of representing a program using transition systems is that the program can be executed symbolically in a simple manner. Moreover, as this representation is general enough, retargeting (e.g., to different types of applications) is just the matter of compilation to the designated transition systems.

Definition 1 (Symbolic State). *A* symbolic state *s is a triple $\langle \ell, \sigma, \Pi \rangle$, where $\ell \in \mathcal{L}$ corresponds to the* concrete *current* program point*, the* symbolic store *σ is a function from program variables to terms over input symbolic variables, and the* path condition *Π is a first-order logic formula over the symbolic inputs which accumulates constraints the inputs must satisfy in order for an execution to follow the corresponding path.* □

Let $s_0 \equiv \langle \ell_0, \sigma_0, \Pi_0 \rangle$ denote the unique *initial* symbolic state. At s_0 each program variable is initialized to a fresh input symbolic variable. For every state $s \equiv \langle \ell, \sigma, \Pi \rangle$, the *evaluation* $[\![e]\!]_\sigma$ of an arithmetic expression e in a store σ is defined as usual: $[\![v]\!]_\sigma = \sigma(v)$, $[\![n]\!]_\sigma = n$, $[\![e + e']\!]_\sigma = [\![e]\!]_\sigma + [\![e']\!]_\sigma$, $[\![e - e']\!]_\sigma = [\![e]\!]_\sigma - [\![e']\!]_\sigma$, etc. The evaluation of conditional expression $[\![c]\!]_\sigma$ can be defined analogously. The set of first-order logic formulas and symbolic states are denoted by FO and $SymStates$, respectively.

Definition 2 (Transition Step). *Given a transition system $\langle \mathcal{L}, \ell_0, \longrightarrow \rangle$ and a state $s \equiv \langle \ell, \sigma, \Pi \rangle \in \mathrm{SymStates}$, the symbolic execution of transition $t : \ell \xrightarrow{\mathsf{op}} \ell'$ returns another symbolic state s' defined as:*

$$s' \triangleq \begin{cases} \langle \ell', \sigma, \Pi \rangle & \text{if } \mathsf{op} \equiv \mathsf{void} \\ \langle \ell', \sigma, \Pi \wedge [\![c]\!]_\sigma \rangle & \text{if } \mathsf{op} \equiv \mathsf{assume(c)} \\ \langle \ell', \sigma[x \mapsto [\![e]\!]_\sigma], \Pi \rangle & \text{if } \mathsf{op} \equiv \mathsf{x := e} \end{cases} \tag{2.1}$$

□

Abusing notation, the execution step from s to s', taking the transition $t : \ell \xrightarrow{\mathsf{op}} \ell'$, is denoted as $s \xrightarrow{t} s'$. Given a symbolic state $s \equiv \langle \ell, \sigma, \Pi \rangle$ we also define $[\![s]\!]$ to be the formula $(\bigwedge_{v \in Vars} v = [\![v]\!]_\sigma) \wedge \Pi$ where *Vars* is the set of program variables. For convenience, when there is no ambiguity, we just refer to the symbolic state s using the pair $\langle \ell, [\![s]\!] \rangle$, where $[\![s]\!]$ is the *constraint* component of the symbolic state s. When we are *not* interested in the program point components of states, we just write $[\![s']\!] \equiv \mathsf{exec}([\![s]\!], \mathsf{op})$ to denote the execution step from s to s'.

A *symbolic path* $\theta \equiv s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_m$ is a sequence of symbolic states such that $\forall i \bullet 1 \leq i \leq m$ the state s_i is a *successor* of s_{i-1}. A symbolic state $s' \equiv \langle \ell', \cdot, \cdot \rangle$ is a successor of another $s \equiv \langle \ell, \cdot, \cdot \rangle$ if there exists a transition relation $\ell \xrightarrow{\mathsf{op}} \ell'$. A path $\theta \equiv s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_m$ is *feasible* if $s_m \equiv \langle \ell_m, \sigma, \Pi \rangle$ such that $[\![\Pi]\!]_\sigma$ is satisfiable. Otherwise, if $[\![\Pi]\!]_\sigma$ is unsatisfiable the path is called *infeasible* and s_m is called an *infeasible* state. Note that in traditional symbolic execution, we do not expand from infeasible states. Here we query a *theorem prover* for satisfiability checking on the path condition. We assume the theorem prover is sound but not complete. That is, the theorem prover must say a formula is unsatisfiable only if it is indeed so.

If $\ell_m \in \mathcal{L}$ and there is no transition from ℓ_m to another program point, then ℓ_m is called the *ending point* of the program. Under that circumstance, if s_m is feasible then s_m is called *terminal* state. A state $s \equiv \langle \ell, \cdot, \cdot \rangle$ is called *subsumed* if there exists another state $s' \equiv \langle \ell, \cdot, \cdot \rangle$ such that $[\![s]\!] \models [\![s']\!]$. Note that s and s' share the same program point ℓ. If there exists a feasible path $\theta \equiv s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_m$ then, for $(0 \leq i < j \leq m)$, we say s_j is *reachable* from s_i in *(j-i) steps*. We say s'' is reachable from s if it is reachable from s in some number of steps.

A *symbolic execution tree* characterizes the execution paths followed during the symbolic execution of a transition system by triggering Eq. (2.1). The nodes/vertices represent symbolic states and the edges represent transitions between states.

EXAMPLE 2.2 *(Symbolic Execution)*: Refer back to the example in Fig. 2.1. Assume

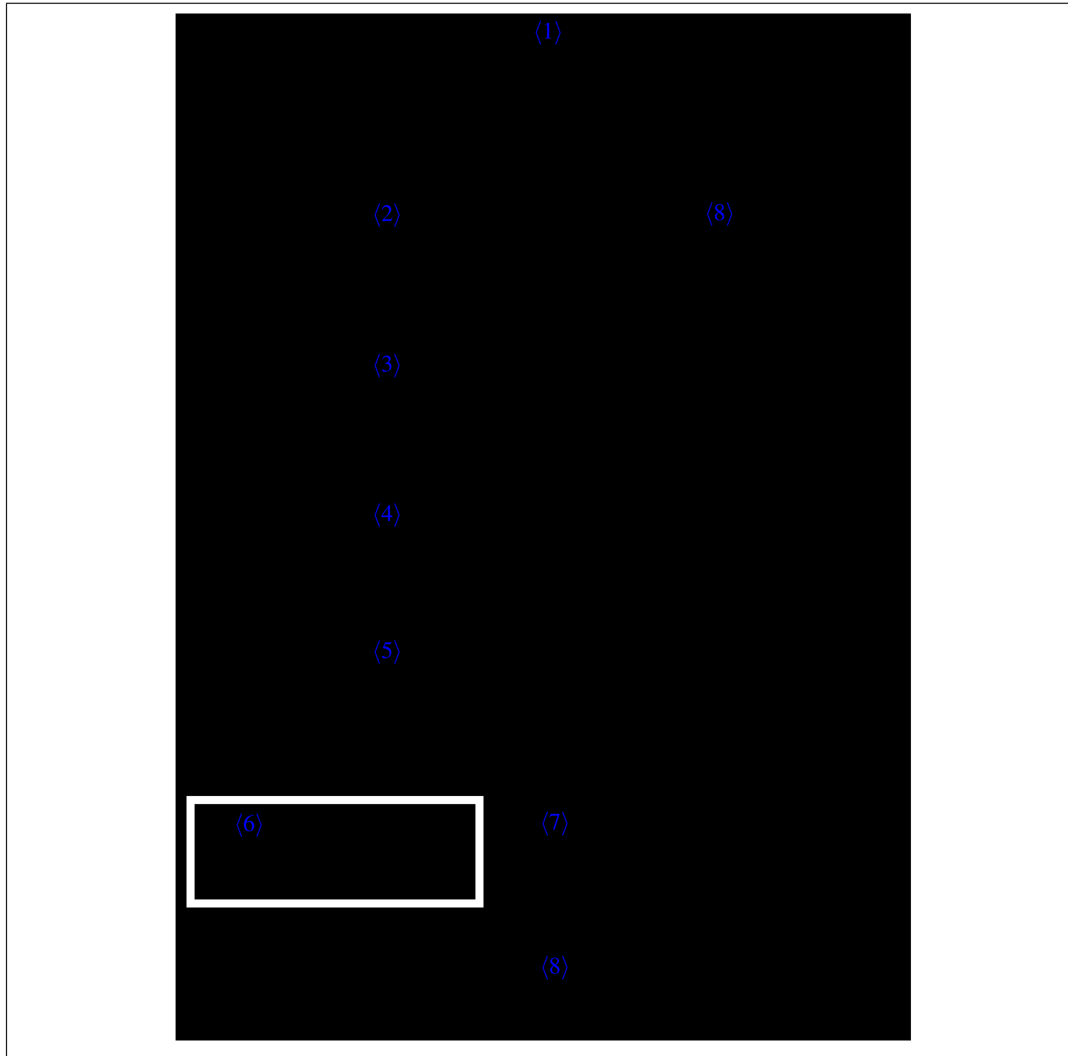


Figure 2.2: Performing Symbolic Execution

that the initial value of variable x is X while the initial value of y is Y. Fig. 2.2 demonstrates the symbolic execution for this program. At the program point $\ell \equiv \langle 6 \rangle$, the path condition $\Pi \equiv X > Y \wedge Y - X > 0$ is unsatisfiable. In other words, the corresponding state is infeasible and requires no further expansion.

2.2 Interpolation

The main approach to counter the path explosion problem in symbolic execution is interpolation [Craig, 1955]. The concept of interpolation has been widely used for verification; recently it has also been adopted in the area of program analysis.

Program Verification via Symbolic Execution

We follow the approach of [Jaffar *et al.*, 2009], where interpolation is in the form of state interpolation (SI). Here our symbolic execution is depicted as a tree rooted at the initial state s_0 and for each state s_i therein, the descendants are just the states obtainable by extending s_i with a feasible transition.

Definition 3 (Safety of A State)**.** *Given a program and a safety property* ψ*, we say a state* $s \in \text{SymStates}$ *is* safe *wrt.* ψ `iff` $[\![s]\!] \models \psi$*.* □

Definition 4 (Safety of A Program)**.** *We say a given program is* safe *wrt. a safety property* ψ *if* $\forall s \in \text{SymStates} \bullet s$ *is reachable from the initial state* s_0 *implies that* s *is safe wrt.* ψ*.* □

Consider one particular feasible path: $s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_2 \cdots s_m$. A *program point* ℓ_i of s_i characterizes a point in the reachability tree in terms of all the remaining possible transitions. Now, this particular path is *safe* wrt. a safety property ψ if for all k, $0 \leq k \leq m$, we have $[\![s_k]\!] \models \psi$. A (state) interpolant at program point ℓ_i, $0 \leq i \leq m$ is simply a set of states S_i containing s_i such that for any state $s'_i \in S_i$, $s'_i \xrightarrow{t_{i+1}} s'_{i+1} \xrightarrow{t_{i+2}} s'_{i+2} \cdots s'_m$, it is also the case that for all k, $i \leq k \leq m$, we have $[\![s'_k]\!] \models \psi$. This interpolant was constructed at program point ℓ_i due to the one path. Consider now all paths from s_0 and with prefix $t_1, \cdots, t_{i-1}$. Compute each of their interpolants. Finally, the interpolant for the subtree (at s_i) of paths just considered is simply the intersection of all the individual interpolants. This notion of interpolant for a subtree provides a weaker notion of *subsumption*. We can now

prune a subtree in case its root is within the interpolant computed for a previously encountered subtree of the same program point.

Definition 5 (Safe Root). *Given a transition system and an initial state s_0, let s be a feasible state reachable from s_0. We say that s is a* safe root *wrt. a safety property ψ, denoted $\triangle_{\psi}(s)$,* `iff` *all states s' reachable from s are safe wrt. ψ.* □

Definition 6 (State Coverage). *Given a transition system and an initial state s_0 and s_i and s_j are two symbolic states such that (1) s_i and s_j are reachable from s_0 and (2) s_i and s_j share the same program point ℓ, we say that s_i* covers *s_j wrt. a safety property ψ, denoted by $s_i \succeq_{\psi} s_j$,* `iff` *$\triangle_{\psi}(s_i)$ implies $\triangle_{\psi}(s_j)$.* □

The impact of state coverage relation is that if (1) s_i and s_j share the same program point ℓ, and (2) s_i covers s_j, and (3) the subtree rooted at s_i has been traversed and proved to be safe, then the traversal of subtree rooted at s_j can be avoided. In other words, we gain performance by *pruning* the subtree at s_j. Obviously, if s_i naturally subsumes s_j, i.e., $[\![s_j]\!] \models [\![s_i]\!]$, then state coverage is trivially achieved. In practice, however, this scenario does not happen often enough.

Let us now introduce the concept of Craig interpolant [Craig, 1955].

Definition 7 (Interpolant). *Given two first-order logic formulas F and G such that $F \models G$, then there exists an* interpolant *H denoted as Intp(F, G), which is a first-order logic formula such that $F \models H$ and $H \models G$, and each variable of H is a variable of both F and G.* □

Definition 8 (Sound Interpolant). *Given a transition system and an initial state s_0, given a safety property ψ and program point ℓ, we say a formula $\overline{\Psi}$ is a* sound interpolant *for ℓ, denoted by $\mathsf{SI}(\ell, \psi)$, if for all state $s \equiv \langle \ell, [\![s]\!] \rangle$ reachable from s_0, $[\![s]\!] \models \overline{\Psi}$ implies that s is a safe root.* □

What we want now is to generate a formula $\overline{\Psi}$ (called *interpolant*), which still preserves the safety of all states reachable from s_i, but is weaker (more general)

than the original formula $[\![s_i]\!]$. In other words, we should have $[\![s_i]\!] \models \mathsf{SI}(\ell, \psi)$. We assume that this condition is always ensured by any implementation of our state-based interpolation. The main purpose of using $\overline{\Psi}$ rather than the original formula associated to the symbolic state s_i is to increase the likelihood of subsumption. That is, the likelihood of having $[\![s_j]\!] \models \overline{\Psi}$ is expected to be much higher than the likelihood of having $[\![s_j]\!] \models [\![s_i]\!]$.

In fact, the perfect interpolant should be the weakest precondition [Dijkstra, 1975] computed for program point ℓ wrt. the transition system and the safety property ψ. We denote this weakest precondition as $\mathsf{wp}(\ell, \psi)$. Any subsequent state $s_j \equiv \langle \ell, [\![s_j]\!] \rangle$ which has $[\![s_j]\!]$ stronger than this weakest precondition can be pruned. However, the weakest precondition, if exists, is too computationally demanding. An interpolant for the state s_i is indeed a formula which approximates the weakest precondition at program point ℓ wrt. the transition system, i.e., $\overline{\Psi} \equiv \mathsf{SI}(\ell, \psi) \equiv \mathit{Intp}([\![s_i]\!], \mathsf{wp}(\ell, \psi))$. A *good* interpolant is one which closely approximates the weakest precondition while can be computed efficiently.

The symbolic execution of a program can be augmented by annotating each program point with its corresponding interpolants such that the interpolants represent the sufficient conditions to preserve the unreachability of any unsafe state. Then, the *basic* notion of pruning with interpolant can be defined as follows.

Definition 9 (Pruning with Interpolant)**.** *Given a symbolic state $s \equiv \langle \ell, [\![s]\!] \rangle$ such that ℓ is annotated with some interpolant $\overline{\Psi}$, we say that s is* pruned *by the interpolant $\overline{\Psi}$ if $[\![s]\!]$ implies $\overline{\Psi}$ (i.e., $[\![s]\!] \models \overline{\Psi}$).* □

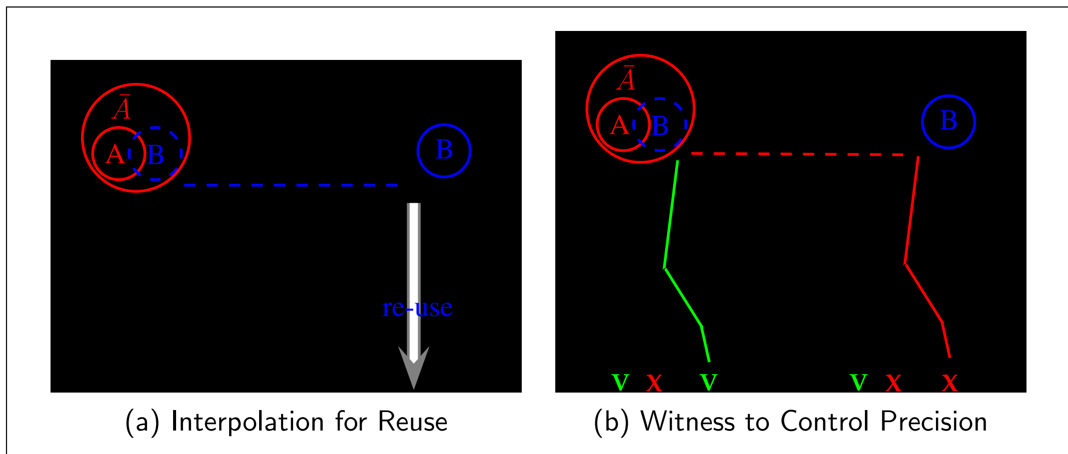
Program Path Analysis via Symbolic Execution

[Jaffar *et al.*, 2008] was the first to introduce the concept of summarization with interpolation. A summarization for a subtree helps reduce the likelihood of fully considering other sub-trees with less general incoming contexts.

For each subtree at node s_i, reuse condition is generated by weakening or gen-

eralizing the context $[\![s_i]\!]$, again by using the concept of *interpolation*. Essentially, we generalize $[\![s_i]\!]$ as long as we preserve the unsatisfiability of all the infeasible paths appeared in the analyzed subtree. The algorithm backtracks and compounds the summarizations computed by the child states and propagates to ancestors for memoing and reuse.

In more details, when a path is analyzed, we extract an interpolant from the formula associated with the symbolic states of its nodes. This is a more general formula stored at each node that preserves all the infeasibility in the path. If one of the nodes is encountered in another path such that its current formula entails the previously computed interpolant and witness, we can avoid exploring the paths from that node. We call this step the *subsumption test*. Whenever the subsumption test fails (i.e., the entailment does not hold), symbolic execution will naturally perform node splitting and duplicate all successors of the node until the next merge point. Alternatively, if the test passes, a node merging is performed. The key insight is that the subsumed node safely *shares* the analysis results of the subsuming node, thus giving rise to the all-important computational optimization of *reuse*.



(a) Interpolation for Reuse (b) Witness to Control Precision

Figure 2.3: Interpolation and Witness for Analysis

In Fig. 2.3(a) we assume that A and B are contexts associated to two sibling subtrees, i.e., the nodes associate to a same program point. For brevity, we will refer to

these subtrees as subtree A and subtree B. W.l.o.g., assume that we have finished analyzing subtree A. In general the two subtrees possess lots of similarities and we want to opportunistically avoid a full exploration of B. In Fig. 2.3(a), context B is not subsumed by context A. However, using the concept of interpolation, context B is subsumed by interpolant $\bar{A}$, a generalization of context A. It means that solutions computed in subtree A can be safely reused in B. We gain performance since, in general, reusing is less costly than fully exploring subtree B.

The use of summarization with interpolation to avoid full path enumeration is *sound*, since to-be-avoided subtrees do not contradict the analysis result already computed for the original (to-be-reused) subtree. However, the original subtree may contain far more paths than the (to-be-avoided) subtree with a less general context. That is, the summarized result might come from a representative path[1] which may now be infeasible in the less general context. Therefore, though sound, the use of interpolation may not guarantee the accuracy level we desire. For illustration, see Fig. 2.3(b), where paths ending with crosses are infeasible. Though context B is subsumed by interpolant $\bar{A}$, reuse should not happen as the representative path for A is no longer feasible in B.

To remedy that, [Jaffar *et al.*, 2008] also introduces the concept of *witness path*. Essentially, when a subtree of paths is analyzed, we also keep the *witness*, a sequence of operations executed by the representative path of the subtree. Assume that we analyze the subtree rooted at node s_i (having the context $[\![s_i]\!]$) and get back the analysis result of which the representative path is $\theta \equiv s_i \xrightarrow{t_{i_1}:\mathsf{op}_{i_1}} \cdots \xrightarrow{t_{i_k}:\mathsf{op}_{i_k}} s_{i_k}$. Our algorithm keeps track of the suffix representative path originated from node s_i, the sequence of operations $\omega_i = \mathsf{op}_{i_1} \wedge ... \wedge \mathsf{op}_{i_k}$. We will call ω_i a witness path of the subtree rooted at node s_i. A new node s_j such that s_i and s_j share the same program point will not be further expanded if: (1) its incoming context $[\![s_j]\!]$ is less general than a previously computed interpolant $\overline{\Psi}_i$ of $[\![s_i]\!]$, i.e., $[\![s_j]\!] \models \overline{\Psi}_i$; and (2) the new

[1] In general, there could be more than one representative paths which contribute to the analysis result. For simplicity, here we assume one only.

context demonstrates that the witness path holds, i.e., $\mathsf{exec}([\![s_j]\!], \omega_i)$ is satisfiable. Otherwise, we say that node s_j cannot be *covered* and a new expansion for that node is required. In a loop-free program, witness path ensures that we achieve *exact* analysis.

The symbolic execution of a program now can be augmented by annotating each program point with its corresponding summarizations. Each summarization contains an interpolant which represents the sufficient condition to preserve all the infeasible paths, an analysis result γ witnessed by the witness ω. Then, the *basic* notion of reuse with interpolant and witness can be defined as follows.

Definition 10 (Reuse with Interpolant and Witness)**.** *Given a symbolic state* $s \equiv \langle \ell, [\![s]\!] \rangle$ *such that* ℓ *is annotated with a summarization* $\langle \overline{\Psi}, \gamma, \omega \rangle$*, we say the result* γ *can be* re-used *at* s *if:*

1. $[\![s]\!] \models \overline{\Psi}$*; and*

2. $\mathsf{exec}([\![s]\!], \omega)$ *is satisfiable.* □

In our symbolic execution framework implemented using CLP($\mathcal{R}$), we represent witnesses as constraint formulas. These representations can be made efficient by using CLP($\mathcal{R}$) projection [Jaffar *et al.*, 1993], which we will briefly discuss in Chapter 3.

Part I

Program Path Analysis

Chapter 3

Loop Unrolling

> Above all, I craved to seize the whole essence, . . . , of some situation that was in the process of unrolling itself before my eyes.
>
> Henri Cartier-Bresson

Programs use *limited* physical resources. Thus determining an upper bound on resource usage by a program is often a critical need. In practice, it should be possible for an experienced programmer, given him/her enough amount of time, to extrapolate from the source code of a well-written program to its *asymptotic* worst-case behavior. But it is often insufficient to just determine the asymptotic behavior of programs.

"*Concrete* worst-case bounds are particularly necessary in the development of embedded systems and hard real-time systems." [Hoffmann *et al.*, 2011]. In other words, a *sound* and *precise* estimation of the resource consumption, for a specific input and under a specific hardware platform, is often required. In this Chapter, we focus on *static* estimation of the Worst-Case Execution Time (WCET). The computed bounds allow safe schedulability analysis of *hard* real-time system. Static methods emphasize *safety* by producing bounds on the execution time, guaranteeing that the

execution time will not exceed these bounds.

A main issue in WCET analysis is to *avoid pessimism* while being *safe* in timing evaluation. Ideally, WCET estimation method should, given an input program, produce a *tight estimate* of the upper-bound of the actual WCET. But first, we need a *timing model* of the hardware platform, in order to come up with the worst-case timing for each basic block in the CFG. This is usually referred to as the problem of low-level analysis. Micro-architectural modeling for low-level analysis is non-trivial and consequently it is almost impossible to achieve exact WCET estimates in CPU cycles. Second, it is crucial to estimate accurately bounds for loops and eliminate *infeasible* paths from bound calculation, especially in the presence of nested loops. This can be partially addressed by requiring user-provided annotations about infeasible paths and loop bounds. Such annotations are usually referred to as *user assertions*. Apart from considerable effort and error-proneness, sometimes the user may not actually know such information. As such, for practicality, the provision of assertions should be *optional*, rather than *mandatory*. A more attractive solution is to automatically detect infeasible paths and derive loop bounds through static *path analysis* methods [Altenbernd, 1996; Ermedahl and Gustafsson, 1997; Gustafsson *et al.*, 2005; Gustafsson *et al.*, 2006].

Path analysis in general is performed separately from low-level analysis. [Theiling *et al.*, 2000], though of which path analysis is not fully automated, emphasizes that precise WCET prediction can be achieved by doing low-level analysis and path analysis separately. As a matter of fact, our path analysis is performed *separately* from low-level analysis. It is intended to be combined with some low-level analysis (e.g., [Theiling *et al.*, 2000]), which gives a worst-case timing for each basic block.

When path analysis is performed separately from low-level analysis, a key issue is the *aggregation phase*, lifting basic block timings (returned by some low-level analysis) to the global timing. At this phase, the information about infeasible paths and loop bounds is crucial because it allows us to exclude certain accumulations of

```
for (i=0; i < n-1; i++)
    for (j=0; j < n-1-i; j++) {
        /* test_and_swap */
}
```
(a) Ex: bubblesort

```
for (i=0; i < n; i++)
    for (j=0; j < n-i; j++) {
        /* do_something */
        i++;
    }
```
(b) Ex: amortized loop

```
for (i=0; i < 10; i++) {
    if (i==4) { /* a */ }
    /* b */
}
```
(c) Ex: down-sampling code

```
while (n > 1) {
    if (n % 2 == 0)
        n = n/2;
    else n = 3*n+1;
}
```
(d) Ex: collatz

```
if (input == 0) {
    /* do_a */
} else { /* do_b */ }
```
(e) Ex: propagation of input

```
if (E < 0) {cond = 0;} /* a */
else {cond = 1;} /* b */
if (cond) result = x/y; /* c */
else result = y; /* d */
```
(f) Ex: mutually exclusive paths

Figure 3.1: Challenging Program Patterns

basic block timings which do not correspond to valid paths. Our work adopts *symbolic simulation* with *loop unrolling* for *automatic* and *precise* detection of infeasible paths and loop bounds.

Infeasible path detection concerns path-sensitivity: without it, accuracy is seriously hampered; but with it, how do we make any algorithm scale given the subsequent explosion in the search space of the symbolic execution? For instance, in Fig. 3.1(e), the WCET of a piece of code depends on the values of its input variables. The fact of whether an analyzer can capture no/partial/full information about the input variables might heavily affect its timing prediction. Similarly, in Fig. 3.1(f), the paths (a,c) and (b,d) are mutually exclusive. Excluding those paths from bound calculation might increase the analysis precision significantly [Altenbernd, 1996]. One trivial example is when the timing of a dominates the timing of b, while at the same time the timing of c dominates the timing of d.

We next discuss the inherent difficulties posed by complicated loops. Scalability is discussed in the later Sections. Here we simply point out some technical aspects of programs that *exacerbate* the already difficult problem.

- *Non-rectangular loops*: we often see triangular loops in sorting algorithms. Fig. 3.1(a) shows bubblesort program. The number of iterations of the inner loop is dependent on the specific iteration of the outer loop. In bounding the total number of the inner loop iterations in this program, general techniques working on parametric bounds would happily accept n^2 as a good bound. Nonetheless, we target the exact bound $n(n-1)/2$ for each known value of n.

- *Amortized loops* [Gulwani and Zuleger, 2010]: in Fig. 3.1(b), the outer loop counter being manipulated inside the inner loop makes it hard to give a tight bound (*linear* instead of *quadratic*).

- *Down-sampling code*: predicting accurately the loop timing is hard if one part of its body is executed less often than the rest of the body (Fig. 3.1(c)). When the timing for /* a */ is significantly larger than the timing for /* b */, the amount of overestimation might become unacceptable.

- *Closed-form is not always possible*: a WCET analysis can produce symbolic expressions which are solved (closed-form) by using off-the-shelf Computational Algebraic Systems (CAS). However, to obtain a closed-form can be unrealistic [Vivancos *et al.*, 2001], as the loop counter can be manipulated nondeterministically in each iteration. An extreme example is the famous Collatz problem in Fig. 3.1(d) [Collatz, 1937]. It is desirable that a WCET analyzer still returns something *safe* for a terminating program (e.g., Collatz problem with a known value of n), even when its closed-form cannot be deduced.

3.1 Contributions and Related Work

To the best of our knowledge, our work is the first fully automated general path analysis method which attempts path-sensitivity and is able to *discover* and *prove* tight upper bound of a resource variable, even in the presence of complicated patterns such as non-rectangular and amortized loops, and down-sampling code even when a closed-form cannot be obtained by traditional CAS. By *prove* here we mean that all infeasible paths detected and used in our analysis are checked by the underlying theorem prover. In the end, we produce not only a bound but also a proof tree so that a third party verifier can certify that the result is *safe*.

Our method is *brute-force* as loops are unrolled. It is different from traditional abstract interpretation (AI) [Cousot and Cousot, 1977] methods dealing with bounds in a way that it never attempts to discover invariants for loops. Instead, we ensure *constraints which are not modified in divergent ways* can be propagated and preserved through loops. Specifically, *variant effects* caused by the loop bodies are abstracted and summarized using a *polyhedral domain* [Cousot and Halbwachs, 1978]. It turns out that this approach is very successful in maintaining flow information stretching across loop-nesting levels and between different loops. The reason is that, though a loop can be complicated, variant effects from different paths in the loop body to variables affecting the control flow of the program, usually *agree* upon *one* abstract value. Thus abstraction is not lossy and crucial flow information can be captured precisely. Experimental results show that, very often, we can come up with not only the exact timing for a benchmark, but also its exact ending context (or its best approximation wrt. the abstract domain used).

A significant work on WCET analysis employing symbolic simulation is done in [Lundqvist and Stenström, 1999]. There low-level analysis and path analysis are combined in one integrated phase. However, that approach has several problems. First, it can only cope with a very simple abstract domain. This leads to limitations in detection of infeasible paths. Second, for the same reason, the approach has

a termination issue with some common programming patterns (see the discussion in [Lundqvist and Stenström, 1999]). Finally, the analysis time is always at least proportional to the actual execution time of the input program. "It leads to a very long analysis since simulation is typically orders of magnitudes slower than native execution" [Wilhelm *et al.*, 2008].

Indeed, exhaustive symbolic execution is very expensive because of both the breadth and depth of the resulting tree. To address the breadth issue, one requires a notion of merge. The analysis precision is then heavily affected by the power of the employed abstract domain. For example, the works [Ermedahl and Gustafsson, 1997; Gustafsson *et al.*, 2005; Gustafsson *et al.*, 2006] employ a form of abstract execution, essentially a combination of symbolic execution and abstract interpretation, using the interval domain. By using the most accurate setting in its AI framework, the method performs full path enumeration and does not scale. To make it practical, similar to [Lundqvist and Stenström, 1999], path-merging is introduced at different levels. We now briefly mention *our merits* in avoiding full path enumeration while attempting path sensitivity.

Our method first addresses the breadth issue using *compounded summarization*. For a loop-free program, we guarantee to produce the *exact* bound while avoiding full path enumeration. For programs with loops, we introduce path-merging *only* at the end of each loop body. However, we employ a more powerful abstract domain, i.e., the polyhedral domain. This obviously results in tighter loop bounds and better detection of infeasible paths. Consequently, our path analysis will be more precise than path analysis performed by [Lundqvist and Stenström, 1999; Ermedahl and Gustafsson, 1997; Gustafsson *et al.*, 2005; Gustafsson *et al.*, 2006]. Another enhancement due to the use of the polyhedral domain is that we do not have any termination issue with common programming practices.

Now consider the depth issue, and this is most affected by loop unrolling. Clearly, analysis must be at least proportional to a concrete execution trace of the pro-

gram [Wilhelm *et al.*, 2008]. For example, the number of states visited by simulating a single-path[1] *quadratic* program will be at least of *quadratic* complexity. [Lundqvist and Stenström, 1999] essentially symbolically executes a fixed number of paths. Thus its performance is mainly determined by the length of the longest path. Even so, that technique does not scale. Similarly, [Ermedahl and Gustafsson, 1997; Gustafsson *et al.*, 2005; Gustafsson *et al.*, 2006] do not address the depth issue. In contrast, we address this fundamental challenge, but now, using *vertically* compounded summarizations. This results in a behavior which we call *depth-wise loop compression*. This is *innovative*. It gives rise for the simulation to be reduced to *linear* complexity for some highly nested loops, even though those loops' complexities are of much higher order. For instance, we can derive the exact bound of bubblesort, a *quadratic* program, in a *linear* number of steps. For further discussion later, we make the following definitions.

Definition 11 (Size Parameter of A Program)**.** *For each program such that its asymptotic time complexity can be expressed in terms of a single variable, indicating the size of the program instance, that single variable is called the* size parameter[2] *of the program.* □

Definition 12 (Reduced to Linear Complexity)**.** *We say that our path analysis on program P is* reduced to linear complexity *if, in terms of a size parameter n, (1) the number of states symbolically executed in the analysis is $O(n)$, whereas (2) the time complexity of P is worse than $O(n)$.* □

Our method naturally supports compositional reasoning, which makes it scale well. Large programs can now be easily split up into a number of smaller programs and the analyzing process can be done in a pipelined manner. In the case that continuation/ending context of a program fragment is captured precisely, we do not compromise the accuracy of the analyses for subsequent fragments.

[1] Every conditional branch is deterministic.

[2] For instance, in sorting algorithms, it is the size of the input array.

Unlike recent methods [Hoffmann *et al.*, 2011; Gulwani and Zuleger, 2010; Bygde *et al.*, 2009], we do not infer parametric bounds for programs. In fact, the outputs we produce are concrete bounds and our method only successfully returns a bound for program on which the symbolic execution terminates. However, by sticking to concrete bounds, we can handle a bigger class of programs and also have the opportunities to discover *tighter* (often *exact*) bounds.

Finally, we again mention the work [Jaffar *et al.*, 2008] from which some conceptual ideas of this Chapter were originated. There the authors address the *resource-constrained shortest path* (RCSP) problem, which is simpler (though NP-hard) than WCET. In RCSP, the cost of traveling from one node to another in a weighted graph, subject to path feasibility determined by some bounds on the resources consumed while traveling, is minimized. The paper introduces the use of interpolation and witnesses for the RCSP problem, but is limited to *loop-free* programs. Furthermore, in RCSP setting, witness path testing can simply be done by recording the amount of resources consumed by the witness, and checking that the adding of the amount to the current consumption does not result in bound violation. In this Chapter, the corresponding problem is far harder.

3.2 Path Analysis vs. Timing Model

Path analysis in general is performed separately from low-level analysis. As a matter of fact, the work [Theiling *et al.*, 2000], though of which path analysis is not fully automated, emphasizes that precise WCET prediction can be achieved by doing low-level analysis and path analysis separately.

When performed separately from with path analysis, low-level analysis often works on control flow graphs (CFG), containing basic blocks as nodes, and on call graphs (CG). The result of this phase is a worst-case execution time for each basic block of the program under examination.

The contribution of a basic block to the timing of a program may vary widely

depending on the execution history. Therefore, by considering timings for basic blocks in their *control flow context*, precision of low-level analysis can be significantly increased. Due to context distinction, nodes in the original graphs are potentially transformed into several nodes in the analysis graphs (called the *extended* CFG). For instance, in [Theiling *et al.*, 2000], contexts indicate through which sequence of function calls and loop iterations control arrives at a basic block. More context distinction offers more accurate bounds. However, for scalability reason, in all WCET tools available, the maximum number of different contexts for each basic block is bounded statically. We define this scenario as *static timing model*.

Definition 13 (Static Timing Model)**.** *A low-level analysis is said to follow* the static timing model (STM) `iff` *for each basic block in the original* CFG *of each program under examination, the maximum number of different contexts , therefore different timings, can be* bounded statically *by a constant which must not depend on the* size parameter *of the input program.* □

The dynamic timing model (DTM) can be easily defined as opposed to static timing model. The work [Lundqvist and Stenström, 1999] indeed follows DTM. In theory, DTM offers precise WCET prediction at the cost of scalability. This is because DTM in general requires low-level analysis and path analysis being *coupled* together.

On the contrary, WCET tools of which the low-level analysis follows STM have a better chance to scale. Works performing *persistent analysis* to capture the behaviors of loops, e.g., [Theiling *et al.*, 2000; Huynh *et al.*, 2011], actually follow the static timing model.

In summary for this discussion, while analysis following the dynamic timing model will in principle produce more accurate results, e.g., [Lundqvist and Stenström, 1999], its effective use is presently beyond reach; there is no scalable analysis using the dynamic timing model yet. Thus our work is designed in such a way that it can be combined with any low-level analysis, which follows the static timing model,

to produce safe WCET bounds.

3.3 Overview

We formulate the WCET path analysis of a program over a symbolic execution tree where each path of the tree is a succession of nodes, each associated with a program point in the program. Each edge is labeled with a statement executed in the corresponding symbolic execution step. *In practice, each statement here is replaced by a distinct basic block in the extended CFGs.* As usual, the initial state is s_0. The context $[\![s_0]\!]$ represents the knowledge about the input of the program. Due to conditional branches and loops, we may come to a same program point but with different contexts. Our method performs depth-first traversal, terminating each path at a state s whenever we are at an ending point of the program, or when $[\![s]\!]$ is unsatisfiable, i.e., an *infeasible* path is detected. In either case, the algorithm records certain information about $[\![s]\!]$ — as a form of learning — and backtracks to the next path.

Multiple contexts allow us to tighten our WCET estimation and prune out unnecessary traversal due to the exclusion of infeasible paths. Unfortunately, a simple enumeration of all contexts is still *exponential*. In the presence of loops and nested loops, it is even worse (e.g., a simple unnested loop of 100 iterations with just *one* conditional branch in its body results in 2^{100} contexts at the end). Not only employs the concept of *reuse with interpolation and witness*, our algorithm also performs *iteration abstraction* at the end of each loop body. Essentially, every iteration of a loop is analyzed as a separate subtree, where end points of the loop body are treated as terminal points of paths. Similar to [Lundqvist and Stenström, 1999; Gustafsson *et al.*, 2005], we reduce the breadth of the symbolic tree by merging paths[3]. This produces only *one* continuation context for analysis of subsequent program fragment.

[3]Though we make use of polyhedral domain.

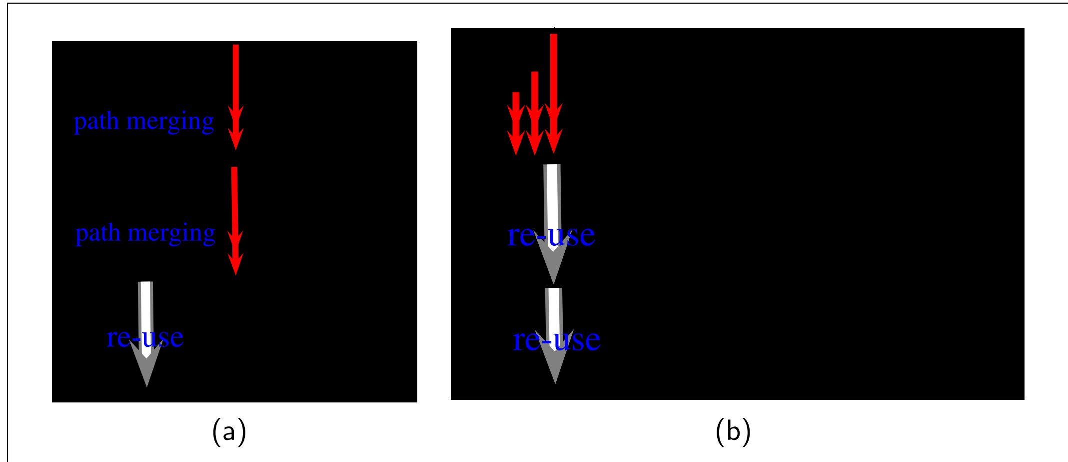


Figure 3.2: Iteration Abstraction and Summarizations of Loop

Furthermore, upon finishing the symbolic subtree of an iteration, we compute its summarization. This gives rise for reuse to happen in a vertical manner. That is, analyses for subsequent iterations with similar behaviors can be quickly deduced (Fig. 3.2(a)). More importantly, however, our summarizations for (inner) loop iterations can be combined vertically for later reuse in the outer loop (Fig. 3.2(b)), so that our path analysis can be reduced to linear complexity. We will illustrate more on this in Section 3.5.

Our work can be viewed as an opportunistic method for the application of *dynamic programming*. Though the concepts of interpolation and summarization have already been well studied, we believe that having them to work with the semantics of exhaustive loop unrolling and path merging while attempting path-sensitivity is a significant contribution.

3.4 Preliminaries

For presentation purpose, in this Chapter, we only deal with structured while loops. We assume that for every loop, there is only one entry node, with exactly one guarded entry transition (to go into the body of the loop) and one guarded exit

transition.

The programs now include an additional variable: the timing variable `t`. Here, the variable of interest `t` models the execution time. Note that this variable is always initialized to 0 and the only operation allowed upon it is a *concrete increment*. In practice, our method works on basic blocks and the amount of increment at each point will be given by some *low-level analysis* module (e.g., [Theiling *et al.*, 2000]). The variable `t` is not used in any other way. (The context on `t` is never used to determine any infeasible path.) For simplicity, in some later examples we just assume every transition uniformly increments `t` by 1. The purpose of our path analysis is to compute a *sound* and *precise* bound for `t` at the end of the execution (across all feasible paths of the program).

A WCET tool usually takes in binary program as input. However, it is much easier for us to perform symbolic execution on transition systems. Translation from binaries into transition systems, similar to the CFG reconstruction problem, is a *non-trivial* task. Fortunately, the task becomes trivial by making use of the work [Theiling, 2002]. Consequently, for clarity and simplicity, our path analysis will be presented on C programs and their transition systems.

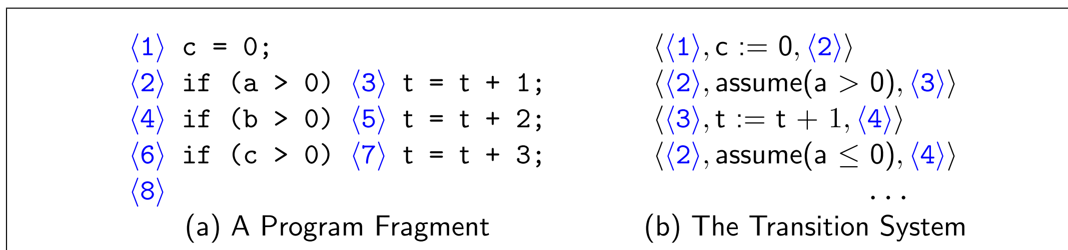


Figure 3.3: From a C Program to its Transition System

EXAMPLE 3.1 : Consider the program fragment in Fig. 3.3(a). The program points are enclosed in angle brackets. Some of the transitions are shown in Fig. 3.3(b). For instance, the transition $\langle\langle 1\rangle, \texttt{c := 0}, \langle 2\rangle\rangle$ represents that the system state switches from program point $\langle 1\rangle$ to $\langle 2\rangle$ executing the operation `c := 0`.

Recall that our transition system is a directed graph. We now introduce concepts

which are required in our loop unrolling framework.

Definition 14 (Loop). *Given a directed graph $G(V, E)$ (our transition system), we call a strongly connected component $S = (V_S, E_S)$ in G with $|E_S| > 0$, a* loop *of G.* □

Definition 15 (Loop Entry). *Given a directed graph $G(V, E)$ and a loop $L = (V_L, E_L)$ of G, we call $\mathcal{E} \in V_L$ a* loop entry *of L, also denoted by $\mathcal{E}(L)$, if no node in V_L, other than $\mathcal{E}$ has a direct predecessor outside L.* □

As we restrict the discussion to structured loops only, indeed, there is no node in a loop, other than the loop entry, having a direct successor outside that loop.

Definition 16 (Ending Point of Loop Body). *Given a directed graph $G(V, E)$, a loop $L = (V_L, E_L)$ of G and its loop entry $\mathcal{E}$. We say a node $u \in V_L$ an* ending point of L's body *if there exists an edge $(u, \mathcal{E}) \in E_L$.* □

We also assume that each loop has only one unique ending point. For each loop, following the back edge from the ending point to the loop entry, we execute a `void` operation. This assumption can be easily achieved by a preprocessing phase.

Definition 17 (Same Nesting Level). *Given a directed graph $G(V, E)$ and a loop $L = (V_L, E_L)$, we say two nodes u and v are in the* same nesting level *if for each loop $L = (V_L, E_L)$ of G, $u \in V_L$* `iff` *$v \in V_L$.* □

Definition 18 (Summarization of a Subtree). *Given two program points ℓ_1 and ℓ_2 such that ℓ_2 post-dominates ℓ_1. Assume that we analyze all the paths from entry point ℓ_1 to exit point ℓ_2 wrt. an incoming context $[\![s]\!]$. The summarization of this subtree is defined as the tuple $[\ell_1, \ell_2, \mathsf{WCET}, \Delta, \overline{\Psi}, \omega]$, where WCET is the worst case timing from ℓ_1 to ℓ_2 and the corresponding path is witnessed by ω,* abstract transformer *Δ is a binary input-output relation between program variables at ℓ_1 and ℓ_2, and* interpolant *$\overline{\Psi}$ is the condition under which this summarization can be safely reused.*

Let wp be the weakest condition such that if we examine the subtree with wp as the incoming context, all infeasible paths (nodes) discovered by previous analysis (using context $[\![s]\!]$) are preserved. As we all know, computing the weakest precondition [Dijkstra, 1975] in general is expensive. The interpolant $\overline{\Psi}$ is indeed an efficiently computable approximation of wp. Specifically, we can define $\overline{\Psi}$ as $Intp(\mathsf{wp}, [\![s]\!])$ (recall the concept of Craig *interpolant* in Chapter 2).

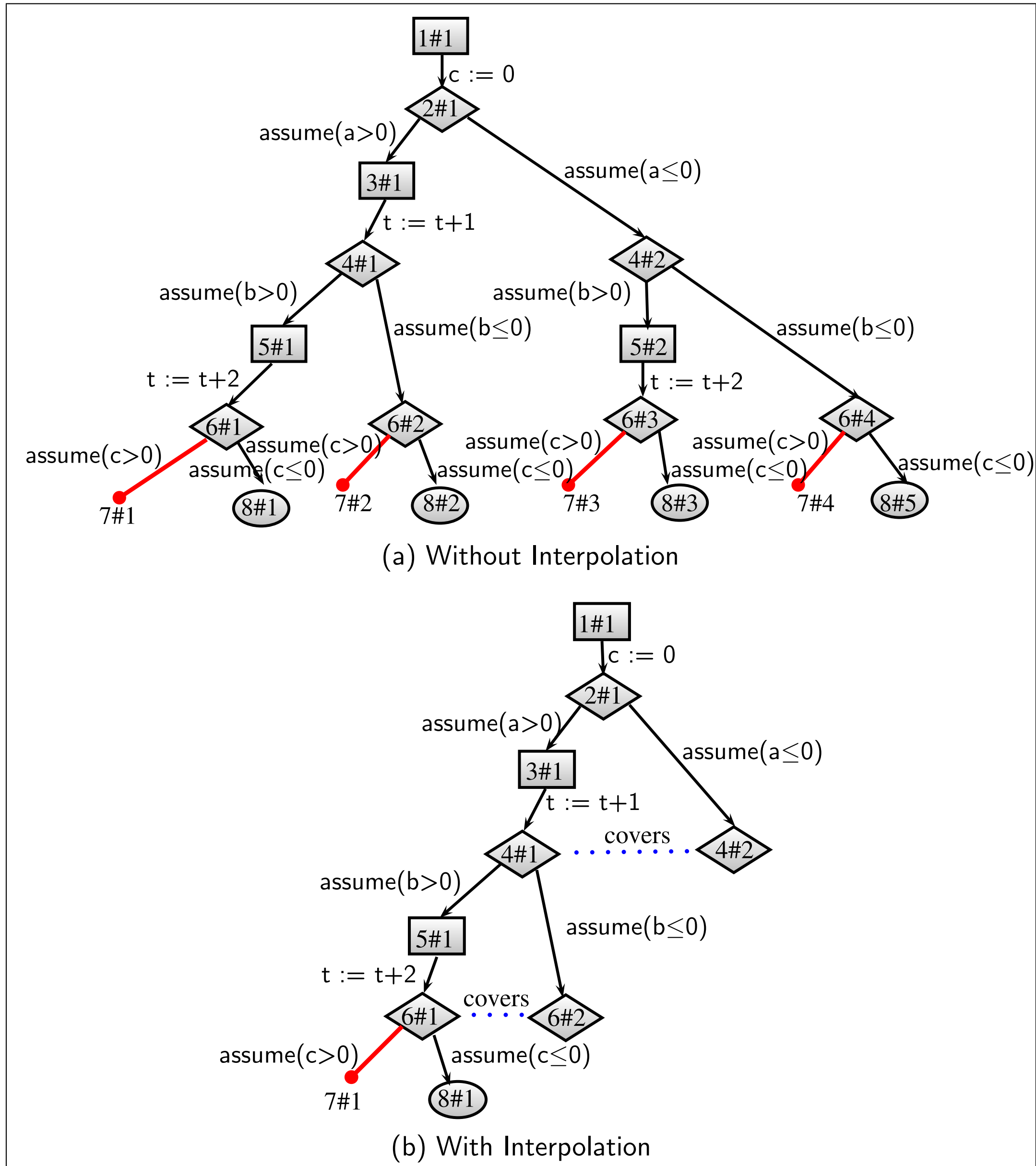
By definition, the abstract transformer Δ will be the abstraction of all feasible paths (wrt. the incoming context $[\![s]\!]$) from ℓ_1 to ℓ_2. Its behavior is similar to a witness. However, as it represents a number of feasible paths, in general, an abstract transformer is *not* a functional relation. We note here that this concept of *abstract transformer* is different from the concept of *abstract transition* developed in [Podelski and Rybalchenko, 2005]. Our abstract transformer is a safe approximation for the input-output relationship of a finite tree, whereas in [Podelski and Rybalchenko, 2005], an abstract transition approximates a path (possibly infinite due to the construction of the closure from the transition relation).

EXAMPLE 3.2 : $\langle 1\rangle$ `if (*) {x++; t++;} else {x+=2; t+=2;}` $\langle 2\rangle$ can be summarized as $[\langle 1\rangle, \langle 2\rangle, 2, \mathsf{x} := \mathsf{x}+1 \vee \mathsf{x} := \mathsf{x}+2, \mathit{true}, \mathsf{x} := \mathsf{x}+2]$. Of course, in practice, we need to avoid the exponential blowup in the size of the abstract transformer. In our implementation built upon CLP($\mathcal{R}$), we use the polyhedral domain for computing the abstract transformer. As a result, the summarization we compute is $[\langle 1\rangle, \langle 2\rangle, 2, x+1 \leq x' \leq x+2, \mathit{true}, x' = x+2]$.

Definition 19 (Summarization of a Program Point)**.** *A summarization of a program point ℓ is the summarization of all paths from ℓ to ℓ' (wrt. the same context), where ℓ' is the nearest program point that post-dominates ℓ s.t. ℓ' is of the same nesting level as ℓ and either is (1) an ending point of the program, or (2) an ending point of some loop body.*

As ℓ' can always be deterministically deduced from ℓ, in the summarization of program ℓ, we usually omit the component about ℓ'.

3.5 Motivating Examples



(a) Without Interpolation

(b) With Interpolation

Figure 3.4: Infeasible Paths in Analyses

EXAMPLE 3.3 *(Infeasible Paths)*: Consider the transition system in Fig. 3.3(b) and its (full) symbolic execution tree in Fig. 3.4(a). Node is labeled `P#C` where `P` is the program point and `C` the context identifier. We label edge by the corresponding

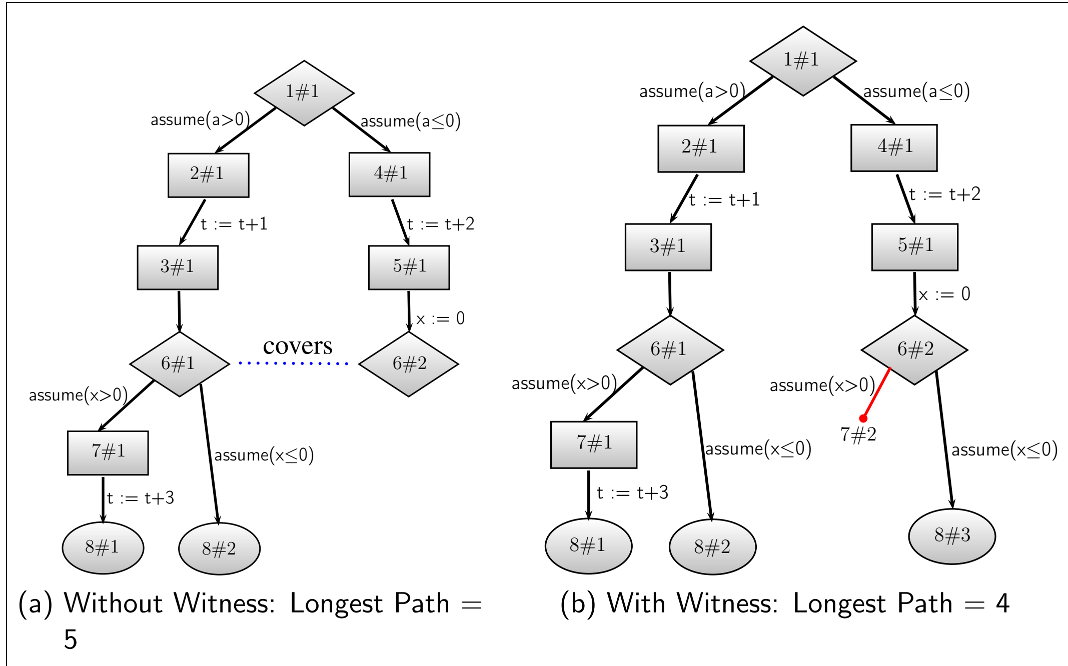
statement. We represent a node before **assume** statements with *diamond*, a node before **assignment** statements with *box*, and terminal node with *ellipse*. *Feasible* transition is denoted by *arrowed* edge, and *infeasible* transition by edge with a *dotted* head.

Without path sensitivity, the path $\langle 1\rangle\langle 2\rangle\langle 3\rangle\langle 4\rangle\langle 5\rangle\langle 6\rangle\langle 7\rangle\langle 8\rangle$, executing the **then** bodies of all the 3 `if` statements, would be considered for bound calculation. This gives 6 as the longest path. However, with path sensitivity as in Fig. 3.4(a), going from program point $\langle 6\rangle$ to program point $\langle 7\rangle$ is not feasible since $c = 0 \wedge c > 0 \equiv false$ (or the path condition at $\langle 7\rangle$ would contain the constraint $0 > 0$). Thus, we infer a tighter bound of 3.

So far, we have illustrated a well-understood benefit of detecting infeasible paths to tighten the estimate. Fig. 3.4(b) depicts a tree computed by our method. The key idea is to generalize the context of each node (if possible) in order to increase the likelihood for reuse, thus we avoid full path enumeration. In this example, our algorithm enlarges the contexts of the nodes `4#1` and `6#1` to the formula $c \leq 0$ since this formula is enough to keep the infeasible path detected at `7#1`. Then, whenever their siblings `4#2` and `6#2` are visited with the contexts $c = 0 \wedge a \leq 0$ and $c = 0 \wedge a > 0 \wedge b \leq 0$, respectively, our algorithm tests that `4#2` and `6#2` are covered by their siblings since those new contexts are less general (i.e., $c = 0 \wedge a \leq 0 \models c \leq 0$ and $c = 0 \wedge a > 0 \wedge b \leq 0 \models c \leq 0$). In Fig. 3.4(b), coverage/reuse is denoted by dashed edge labeled with “covers”.

```
⟨1⟩ if (a > 0)
⟨2⟩     t = t + 1;
⟨3⟩ else {
⟨4⟩     t = t + 2;
⟨5⟩     x = 0;
    }
⟨6⟩ if (x > 0)
⟨7⟩     t = t + 3;
⟨8⟩
```

EXAMPLE 3.4 *(Witness Paths)*: Though covering a node (using interpolant) may reduce the search space while preserving correctness, it does not necessarily preserve *accuracy* of the analysis. Consider the program fragment above. A possible analysis in Fig. 3.5(a). The interpolant associated with the subtree rooted at `6#1` is *true* since there are no infeasible paths. Hence `6#1` covers the context of `6#2`. Using the same reasoning as in previous example, a possible WCET estimate is 5 by considering the path: ⟨1⟩ ⟨4⟩ ⟨5⟩ ⟨6⟩ ⟨7⟩ ⟨8⟩ (note that this path is infeasible though). The estimate is calculated by adding 2 from the transition `4#1` to `5#1` and 3 from transition `7#1` to `8#1`.



(a) Without Witness: Longest Path = 5

(b) With Witness: Longest Path = 4

Figure 3.5: Witnesses Improve Precision

For better precision, we should expand `6#2`, shown in Fig. 3.5(b) The key observation is that the new subtree rooted at `6#2` contains an infeasible path if $x \leq 0$. This infeasible path eliminates the potential path from `6#2` to `7#2` which would have provided a longer (5) but *spurious* answer. Thus we are left with a tighter estimate (4) from the path `1#1`, `2#1`, `3#1`, `6#1`, `7#1`, and `8#1`.

This example illustrates the need to strengthen the condition of coverage for better accuracy. This is done by storing at each subtree, a *witness* ω which concretely represents the WCET path for that very subtree. This witness is then used (in conjunction with the interpolant) to determine coverage/reuse.

In Fig. 3.5(b), the context at node 6#2 is $[\![s_{6\#2}]\!] \equiv a \leq 0 \wedge x = 0$. The interpolant at 6#1 is $\overline{\Psi}_{6\#1} \equiv \mathit{true}$. It is straightforward to see that $[\![s_{6\#2}]\!] \models \overline{\Psi}_{6\#1}$. In addition, we now test if the witness still holds, i.e., we are testing whether $(\omega_{6\#1} \equiv x > 0) \wedge [\![s_{6\#2}]\!]$ is satisfiable. Since $a \leq 0 \wedge x = 0 \wedge x > 0$ is unsatisfiable, the algorithm must explore the node 6#2, thus obtaining a more precise (actually the exact) bound.

EXAMPLE 3.5 *(Superlinear)*: Consider the bubblesort program as the following:

```
⟨0⟩  i = 0, n = 4;
⟨1⟩  while (i < n-1) {
⟨2⟩      j = 0;
⟨3⟩      while (j < n-1-i) {
⟨4⟩          if (*) {
⟨5⟩              /* swap(a, j, j+1) */
             }
⟨6⟩          j++;
⟨7⟩
         }
⟨8⟩      i++;
⟨9⟩
     }
⟨10⟩
```

The analysis is in Fig. 3.6. We represent a separate computation for an iteration as a rectangle with double boundaries. Each when summarized and memoed will be replaced by a single abstract transition denoted as a double-headed arrow. Reuse of summarization is denoted as a double-bodied arrow with the program point and previously encountered context attached. For space, we also shorten the syntax for our `assume` and `assignment` operations. For simplicity, in this example, every (non-abstract) transition increments the timing variable `t` by 1 (even for `swap` function). Witness paths are also omitted as they do not improve accuracy in this example.

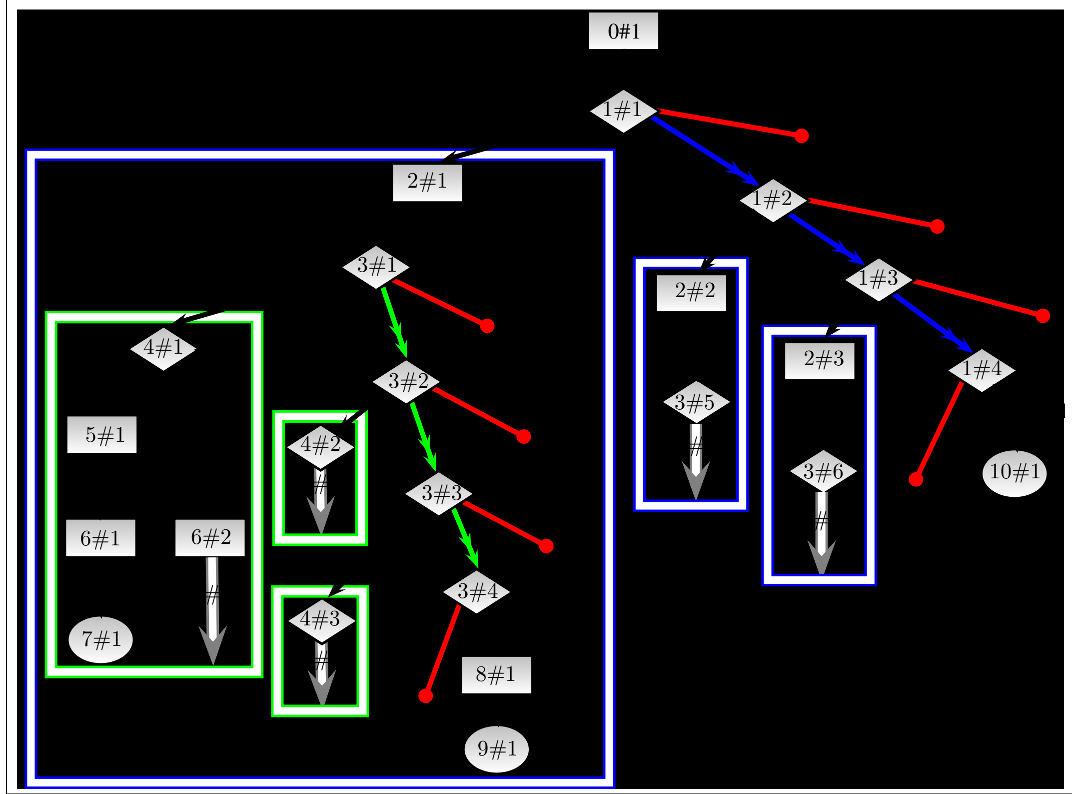


Figure 3.6: Superlinear Analysis of bubblesort

We arrive at `2#1` analyzing the first iteration of the outer loop. From choice point `3#1` we go into the first iteration of the inner loop. The path `4#1 5#1 6#1 7#1` is analyzed normally. The summarizations of program point $\langle 6 \rangle$ and program point $\langle 5 \rangle$ are computed and stored during backtracking. It is worth to note that the summarization for `6#1` is $[\langle 6\#1 \rangle, 1, j' = j + 1, \mathit{true}, \cdot]$ (it is implicitly understood that $i' = i \wedge n' = n$). In the next visit of $\langle 6 \rangle$, which is `6#2`, we obviously can make use of that summarization.

The summarization for `4#1` then is computed as $[\langle 6\#1 \rangle, 3, j' = j + 1, \mathit{true}, \cdot]$. The WCET is the maximum increment for the timing variable `t` by considering both paths originated from `4#1`. The interpolant simply is *true* as there are no infeasible paths. The abstract transformer is combined from the two paths, which is $(\mathtt{swap}(a, j, j + 1) \wedge j' = j + 1) \vee (j' = j + 1)$. After simplification (here we ignore

the effects on array a), it yields just $j' = j + 1$. The whole iteration is then replaced by a single transition (double headed arrow) from `3#1` to `3#2`, making use of the abstract transformer $\Delta \equiv j < n - 1 - i \wedge j' = j + 1$ (note that the loop entry condition is included). We continue the analysis of `3#2` with *one* abstract context, $\langle\langle 3\#2\rangle, i = 0 \wedge j = 1 \wedge n = 4\rangle$. Similarly, we go into the body of the inner loop at `4#2` and `4#3`, making use of the previous summarization for `4#1` to continue the analysis. At `3#4`, the attempt going into the inner loop body fails as an infeasible path is detected.

When we backtrack, by treating double-headed arrows as normal transitions, we come up a serialization (4 instances) of compounded summarizations for program point $\langle 3\rangle$:

$[\langle 3\#4\rangle, 3, i' = i + 1, n - 1 - i \leq j, \cdot]$

$[\langle 3\#3\rangle, 7, i' = i + 1 \wedge j' = j + 1, n - 2 - i \leq j < n - 1 - i, \cdot]$

$[\langle 3\#2\rangle, 11, i' = i + 1 \wedge j' = j + 2, n - 3 - i \leq j < n - 2 - i, \cdot]$

$[\langle 3\#1\rangle, 15, i' = i + 1 \wedge j' = j + 3, n - 4 - i \leq j < n - 3 - i, \cdot]$

The abstract transformers for those summarizations are computed in a similar manner as how the abstract transformer for `4#1` is computed. On the other hand, the interpolant for `3#4` is the weakest condition which ensures the attempt re-entering the loop body at `3#4` fails, i.e., the corresponding infeasible path is preserved. The interpolant for `3#3` preserves not only such infeasible path but also the infeasible path on the attempt exiting the loop at `3#3`. Similarly, the process goes on for `3#2` and `3#1`. Utilizing compounded summarization saves us from analyzing the inner loop again in the future exploration of subsequent outer loop's iterations. Specifically, at `3#5`, we reuse the summarization of `3#2`; while, at `3#6`, we reuse the summarization of `3#3`. As a result, even though the complexity and the WCET of bubblesort program is *quadratic* to n, the number of the inner loop's iterations explored by our method is just *linear* to n. This behavior remains (for similar programs) even when we introduce more nesting levels. This fact sets

us apart from other typical simulation approaches (e.g., [Gustafsson *et al.*, 2005; Lundqvist and Stenström, 1999]).

3.6 Symbolic Simulation Algorithm

function $\mathsf{SS}(s, \mathcal{P})$
Let s be $\langle \ell, [\![s]\!] \rangle$
$\langle 1 \rangle$ **if** $([\![s]\!] \equiv \mathit{false})$ **return** $[\ell, -\infty, \mathit{false}, \mathit{false}, \mathit{false}]$
$\langle 2 \rangle$ **if** $(\mathsf{outgoing}(\ell, \mathcal{P}) \equiv \emptyset)$ **return** $[\ell, 0, \mathsf{Id}(\tilde{x}, \tilde{x}'), \mathit{true}, \mathit{true}]$
$\langle 3 \rangle$ **if** $(\mathsf{loop_end}(\ell, \mathcal{P}))$ **return** $[\ell, 0, \mathsf{Id}(\tilde{x}, \tilde{x}'), \mathit{true}, \mathit{true}]$
$\langle 4 \rangle$ $\mathrm{S} := \mathsf{memoed}(s)$
$\langle 5 \rangle$ **if** $(\mathrm{S} \not\equiv \mathit{false})$ **return** S **endif**
$\langle 6 \rangle$ **if** $(\mathsf{loop_entry}(\ell, \mathcal{P}))$
$\langle 7 \rangle$ $\quad \mathrm{S}_1 := [\ell, \mathsf{WCET}, \Delta(\tilde{x}, \tilde{x}'), \overline{\Psi}(\tilde{x}), \omega(\tilde{x})]$
$\quad\quad := \mathsf{TransStep}(s, \mathcal{P}, \{\mathsf{entry}(\ell, \mathcal{P})\})$ /* Unroll the first iteration */
$\langle 8 \rangle$ $\quad$ **if** $(\omega(\tilde{x}) \equiv \mathit{false})$
$\langle 9 \rangle$ $\quad\quad \overline{\mathrm{S}} := \mathsf{JoinHorizontal}(\mathrm{S}_1, \mathsf{TransStep}(s, \mathcal{P}, \{\mathsf{exit}(\ell, \mathcal{P})\}))$
$\quad$ **else**
$\langle 10 \rangle$ $\quad\quad s \xrightarrow{\Delta(\tilde{x}, \tilde{x}')} s'$ /* Execute abstract transition $\Delta(\tilde{x}, \tilde{x}')$ */
$\langle 11 \rangle$ $\quad\quad \mathrm{S}_{n-1} := \mathsf{SS}(s', \mathcal{P})$ /* Recursively unroll the loop */
$\langle 12 \rangle$ $\quad\quad \mathrm{S}_n := \mathsf{JoinVertical}(\mathrm{S}_1, \mathrm{S}_{n-1})$
$\langle 13 \rangle$ $\quad\quad \overline{\mathrm{S}} := \mathsf{JoinHorizontal}(\mathrm{S}_n, \mathsf{TransStep}(s, \mathcal{P}, \{\mathsf{exit}(\ell, \mathcal{P})\}))$
$\quad$ **endif**
else
$\langle 14 \rangle$ $\quad \overline{\mathrm{S}} := \mathsf{TransStep}(s, \mathcal{P}, \mathsf{outgoing}(\ell, \mathcal{P}))$
endif
$\langle 15 \rangle$ memo and **return** $\overline{\mathrm{S}}$
end function

Figure 3.7: Symbolic Simulation Algorithm: Main Function

In this Section, our presented algorithm (shown in Fig. 3.7 and 3.8) only deals with loops. Recursive functions can be treated in a similar manner. Our symbolic simulation algorithm manipulates a global memo table, which is initialized to empty. During analysis, new summarizations of the form $[\ell, \mathsf{WCET}, \Delta(\tilde{x}, \tilde{x}'), \overline{\Psi}(\tilde{x}), \omega(\tilde{x})]$ (as in Def. 19) will be inserted into the memo table (line 15). Here we use $\tilde{x}$ to refer the

function JoinVertical$(\mathrm{S}_1, \mathrm{S}_2)$
Let S_1 be $[\ell, \mathsf{WCET}_1, \Delta_1(\tilde{x}, \tilde{x}'), \overline{\Psi}_1(\tilde{x}), \omega_1(\tilde{x})]$
Let S_2 be $[\ell', \mathsf{WCET}_2, \Delta_2(\tilde{x}', \tilde{x}''), \overline{\Psi}_2(\tilde{x}'), \omega_2(\tilde{x}')]$
⟨16⟩ $\mathsf{WCET} := \mathsf{WCET}_1 + \mathsf{WCET}_2$
⟨17⟩ $\Delta(\tilde{x}, \tilde{x}'') := \Delta_1(\tilde{x}, \tilde{x}') \wedge \Delta_2(\tilde{x}', \tilde{x}'')$
⟨18⟩ $\overline{\Psi}(\tilde{x}) := \overline{\Psi}_1(\tilde{x}) \wedge \mathsf{pre}(\Delta_1(\tilde{x}, \tilde{x'}), \overline{\Psi}_2(\tilde{x}'))$
⟨19⟩ $\omega(\tilde{x}) := \omega_1(\tilde{x}) \wedge \Delta_1(\tilde{x}, \tilde{x}') \wedge \omega_2(\tilde{x}')$
⟨20⟩ **return** $[\ell, \mathsf{WCET}, \Delta(\tilde{x}, \tilde{x}''), \overline{\Psi}(\tilde{x}), \omega(\tilde{x})]$
end function

function JoinHorizontal$(\mathrm{S}_1, \mathrm{S}_2)$
Let S_1 be $[\ell, \mathsf{WCET}_1, \Delta_1(\tilde{x}, \tilde{x}'), \overline{\Psi}_1(\tilde{x}), \omega_1(\tilde{x})]$
Let S_2 be $[\ell, \mathsf{WCET}_2, \Delta_2(\tilde{x}, \tilde{x}'), \overline{\Psi}_2(\tilde{x}), \omega_2(\tilde{x})]$
⟨21⟩ **if** $(\mathsf{WCET}_1 \geq \mathsf{WCET}_2)$
⟨22⟩ $\mathsf{WCET} := \mathsf{WCET}_1$
⟨23⟩ $\omega(\tilde{x}) := \omega_1(\tilde{x})$
else
⟨24⟩ $\mathsf{WCET} := \mathsf{WCET}_2$
⟨25⟩ $\omega(\tilde{x}) := \omega_2(\tilde{x})$
endif
⟨26⟩ $\Delta(\tilde{x}, \tilde{x}') := \Delta_1(\tilde{x}, \tilde{x}') \vee \Delta_2(\tilde{x}, \tilde{x}')$
⟨27⟩ $\overline{\Psi}(\tilde{x}) := \overline{\Psi}_1(\tilde{x}) \wedge \overline{\Psi}_2(\tilde{x})$
⟨28⟩ **return** $[\ell, \mathsf{WCET}, \Delta(\tilde{x}, \tilde{x}'), \overline{\Psi}(\tilde{x}), \omega(\tilde{x})]$
end function

function TransStep$(s, \mathcal{P}, \mathit{TransSet})$
Let s be $\langle \ell, [\![s]\!] \rangle$
⟨29⟩ $\overline{\mathrm{S}} := [\ell, 0, \mathit{false}, \mathit{true}, \mathit{true}]$
⟨30⟩ **foreach** $(\mathit{trans} \in \mathit{TransSet} \wedge \mathit{trans}$ contains $\mathsf{t} := \mathsf{t}+\alpha)$ **do**
⟨31⟩ $s \xrightarrow{\mathit{trans}} s'$ /* Execute *trans* */
⟨32⟩ $[\ell', \mathsf{WCET}, \Delta, \overline{\Psi}, \omega] := \mathsf{SS}(s', \mathcal{P})$
⟨33⟩ $\mathrm{S} := [\ell, \mathsf{WCET} + \alpha, \mathsf{combine}(\mathit{trans}, \Delta), \mathsf{pre}(\mathit{trans}, \overline{\Psi}), \mathsf{combine}(\mathit{trans}, \omega)]$
⟨34⟩ $\overline{\mathrm{S}} := \mathsf{JoinHorizontal}(\overline{\mathrm{S}}, \mathrm{S})$
endfor
⟨35⟩ **return** $\overline{\mathrm{S}}$
end function

Figure 3.8: Symbolic Simulation Algorithm: Helper Functions

set of program variables. For better understanding, and also to be closer to our implementation, different versions of $\tilde{x}$ are used to indicate the changes of the program variables by functions/predicates. For example, the abstract transformer $\Delta(\tilde{x}, \tilde{x}')$ indicates the changes of the program variables from $\tilde{x}$ to $\tilde{x}'$ while the interpolant $\overline{\Psi}(\tilde{x})$ indicates a constraint formula on the program variables $\tilde{x}$.

In constructing compounded summarization, we rely on two important functions, namely JoinVertical and JoinHorizontal. Each of them takes in, as inputs, two summarizations S_1 and S_2, respectively summarizing two subtrees T_1 and T_2. T_1 and/or T_2 could well be just a single transition. In fact, even when they are not, we still treat them each as a single *abstract* transition, the abstract transformer plays the role of the transition relation. We first explain these two *crucial* functions. Then, we will discuss our algorithm as a whole. Some implementation details are deferred till Section 3.7.

Compounding Vertically two Summarizations: We achieve this by JoinVertical in Fig. 3.8. JoinVertical summarizes a compounded subtree T, where T_2 suffixes T_1. In other words, a path θ_1 in T_1 followed by a path θ_2 in T_2 corresponds a path θ (possibly infeasible) in T. The WCET of T is computed intuitively (line 16) whereas T's abstract transformer is computed as the conjunction of the abstract transformers of T_1 and T_2 (line 17). Similar for the case of T's witness path (line 19). The only difference is that, T_1's witness and T_2's witness are related by the abstract transformer Δ_1 of T_1. By treating T_1 as an abstract transition, computing the interpolant for T relies on the operation $\mathsf{pre}(\Delta_1(\tilde{x}, \tilde{x}'), \overline{\Psi}_2(\tilde{x}'))$ to produce a formula which under-approximates the *weakest precondition* of the postcondition $\overline{\Psi}_2(\tilde{x}')$ wrt. the transition relation $\Delta_1(\tilde{x}, \tilde{x}')$. That is, approximating the formula $\Delta_1(\tilde{x}, \tilde{x}') \rightarrow \overline{\Psi}_2(\tilde{x}')$ [Bjørner *et al.*, 1997].

Compounding Horizontally two Summarizations: We achieve this by JoinHorizontal in Fig. 3.8. JoinHorizontal summarizes a compounded subtree T, where T_1 and T_2 are siblings. This is the `join` operation that we often see in other techniques [Lundqvist

and Stenström, 1999; Ermedahl and Gustafsson, 1997; Gustafsson *et al.*, 2005]. The compounded WCET and witness are computed intuitively (lines 21-25). Preserving all infeasible paths in T requires preserving infeasible paths in both T_1 and T_2 (line 27). The input-output relationship of T is safely abstracted as the disjunction of the input-output relationships of T_1 and T_2 respectively (line 26).

Inputs and Output: The inputs of our main function, namely SS, include the current symbolic state s and the transition system $\mathcal{P}$ deduced from the original program.

Initially, SS is invoked with the initial state $s_0 \equiv \langle \ell_0, [\![s_0]\!] \rangle$, where $[\![s_0]\!]$ captures the input information of the original program. For a given state s, SS performs a depth-first traversal of the execution tree rooted at s; summarizations are collected in a post-order manner. Its final product is a summarization $[\ell_0, \mathsf{WCET}, \Delta(\tilde{x}, \tilde{x}'), \overline{\Psi}(\tilde{x}), \omega(\tilde{x})]$, representing the whole analyzed program.

When to Reuse: Function memoed checks whether a summarization has already been memoed and can be reused. Specifically, given a symbolic state $s \equiv \langle \ell, [\![s]\!] \rangle$, we use memoed($s$) to test if there is a tuple $\mathrm{S} \equiv [\ell, \mathsf{WCET},\ \Delta(\tilde{x}, \tilde{x}'), \overline{\Psi}(\tilde{x}), \omega(\tilde{x})]$ stored before such that $[\![s]\!] \models \overline{\Psi}(\tilde{x})$ and $\omega(\tilde{x}) \wedge [\![s]\!]$ is satisfiable. If yes, we say that we *reuse* at s and return S. Otherwise, *false* is returned.

Base Cases: Our algorithm is most naturally implemented recursively. The function SS handles four base cases. First, when the context $[\![s]\!]$ is unsatisfiable (line 1), no execution needs to be considered. Note that here the path-sensitivity plays a role since only (provably) executable paths will be considered. Second, the algorithm checks if s is a terminal state (line 2). Here $\mathsf{Id}(\tilde{x}, \tilde{x}') \equiv \forall i \in \{1, ..., |\tilde{x}|\} \bullet \tilde{x}'[i] = \tilde{x}[i]$, i.e., it represents the transition relation for `void` operation. Ending point of a loop is treated similarly in the third base case (line 3). The last base case, lines 4-5, is the case that a summarization can be reused.

Expanding to next Program Points: Line 14 depicts the case when transitions can be taken from the current program point ℓ, and ℓ is not a loop starting point.

Here we call TransStep to move recursively to next program points. The returned value is then passed on. TransStep implements the traversal of transition steps emanating from ℓ by calling SS recursively and then compounds the returned summarizations into a summarization of ℓ. The arguments of TransStep are a state s, the transition system $\mathcal{P}$, and a set of outgoing transitions *TransSet* to be explored.

For each transition in *TransSet*, TransStep extends the current state with the transition. Resulting child state is then given as an argument in a recursive call to SS (line 32). From each summarization of a child returned by the call to SS, the algorithm computes a component summarization, contributed by that particular child to the parent (line 33). All of such components will be compounded using the JoinHorizontal function (line 34).

The interpolant for the child state is propagated back to its parent using the *precondition operation* pre, where $\mathsf{pre}(t, \overline{\Psi})$ denotes the precondition of the postcondition $\overline{\Psi}$ wrt. the transition t. In an ideal case, we would want this operation to return the *weakest precondition*. But in general that might not be affordable. The combine function simply conjoins the corresponding constraints and performs projections to reduce the size of the formula.

Loop Handling with Compounded Summarization: Lines 7-13 handle the case when the current program point is the loop entry point. For simplicity, we assume all loops to be in the form of structured `while` loops, where entry denotes the transition going into the body of the loop, and exit denotes the transition exiting the loop.

Upon encountering a loop, our algorithm attempts to unroll it once by calling procedure TransStep to explore the entry transition (line 7). When the returned witness is *false*, it understands that we cannot go into the loop body anymore, thus proceeds to exit branch. The returned summarization is compounded (using JoinHorizontal) with the summarization of previous unrolling attempt (line 9). On the contrary, if some feasible paths found by going into the loop body, we use the

returned abstract transformer to produce a new context. From this context, we recursively call SS to do the rest of the unrolling process. The returned information is then compounded (using JoinVertical) with the first unrolling attempt and later compounded (using JoinHorizontal) with the analysis of the exit branch (line 10-13). Our algorithm can be *reduced to linear complexity* because these compounded summarizations of the inner loop(s) can be reused in later iteration of the outer loop.

We conclude this Section with a correctness statement. The validity of it follows from the fact that in our framework, in forward computation, we perform only abstraction, by path merging (similarly, computing the abstract transformer) as in any abstract interpretation framework. In backward learning, our computed interpolants are indeed stronger than the corresponding weakest preconditions. This ensures that every reuse is *safe*.

Theorem 1 (Soundness)**.** *Our symbolic simulation algorithm always produces* safe *WCET estimates.*

3.7 Implementation Details

3.7.1 Propagating Witnesses

We refer to line 19 in Fig. 3.8. As shown, witness ($\omega(\tilde{x})$) is constructed from the constraints along the path that gives rise to WCET. Such path can be very long and naively recording it would be a source of inefficiency. Recall that we use witnesses to test for feasibility of a solution within memoed function. That is, given a state $s \equiv \langle \ell, [\![s]\!] \rangle$ and a witness $\omega(\tilde{x})$, we test if $[\![s]\!] \wedge \omega(\tilde{x})$ is satisfiable. In general, the witness $\omega(\tilde{x})$ contains other variables, which are disjoint from the variables of $[\![s]\!]$. However, $[\![s]\!] \wedge \omega(\tilde{x})$ is satisfiable `iff` $[\![s]\!] \wedge (\exists var(\omega) \setminus var([\![s]\!]) \bullet \omega(\tilde{x}))$ is satisfiable. Therefore, rather than maintaining $\omega(\tilde{x})$, we maintain a formula that is equivalent to $\exists var(\omega) \setminus var([\![s]\!]) \bullet \omega(\tilde{x})$. CLP($\mathcal{R}$) projection [Jaffar *et al.*, 1993] is useful here.

3.7.2 Computing the Abstract Transformer

Let us again refer to Fig. 3.8. The operation in line 17 is similar to the manipulation of witness paths and we deal with it similarly (by projection). However, operation in line 26 requires more attention. In fact, we make use of the *polyhedral library* [Cousot and Halbwachs, 1978; Verge, 1994] to handle this disjunction, computed as the *convex hull* of its components. As a result, we only capture linear input-output relationships of program variables. Input-output relationships are in general non-linear. Fortunately, transformations of program variables which affect the flow of the program are very often just linear and are captured precisely by the polyhedral domain.

3.7.3 Computing the Interpolants

Refer to the program fragment in the next page. There are 2 infeasible paths of the program:

$\langle 0\rangle a = 1 \wedge b = 1 \wedge y = -1 \langle 1\rangle x < 0 \langle 2\rangle y' = a \langle 4\rangle y' \leq 0 \langle 6\rangle$

$\langle 0\rangle a = 1 \wedge b = 1 \wedge y = -1 \langle 1\rangle x \geq 0 \langle 3\rangle y' = b \langle 4\rangle y' \leq 0 \langle 6\rangle$

```
⟨0⟩ a=1,b=1,y=-1;
⟨1⟩ if (x<0)
⟨2⟩     y=a;
    else
⟨3⟩     y=b;
⟨4⟩ if (y>0) ⟨5⟩ x=1;
⟨6⟩
```

By infeasibility, the state at $\langle 6\rangle$ for the two paths here is *false*. If we use the notion of weakest precondition [Dijkstra, 1975] to generalize preceding states for the first path we get the weakest precondition $\neg(\exists y' \,.\, x < 0 \wedge y' = a \wedge y' \leq 0) \equiv x < 0 \rightarrow a > 0$ at $\langle 1\rangle$ for the first path, and $\neg(\exists y' \,.\, x >= 0 \wedge y' = b \wedge y' \leq 0) \equiv x \geq 0 \rightarrow b > 0$ for the second path. Our issue is how to approximate the weakest precondition for a path efficiently. Both paths share a prefix $\langle 0\rangle$ $\langle 1\rangle$. The desired weakest precondition

for $\langle 1 \rangle$, which would maintain the infeasibility of both paths, is the conjunction of the weakest preconditions of both paths: $(x < 0 \rightarrow a > 0) \wedge (x \geq 0 \rightarrow b > 0)$ which is a complex formula involving conjunction and disjunction. Combining the approximations of various paths efficiently is another issue. There are two techniques in our system.

Using Constraint Deletion: Given the paths as before, we remove all constraints that are not necessary to ensure infeasibility. To ensure the infeasibility of the first path, we may remove $b = 1$, $y = -1$, and $x < 0$. For the second path, we may remove $a = 1$, $y = -1$ and $x \geq 0$. Here, both paths share the prefix $\langle 0 \rangle$ $\langle 1 \rangle$ which contains $a = 1$, $b = 1$, and $y = -1$. Both paths agree on the removal of $y = -1$, hence we remove it, obtaining the state $a = 1 \wedge b = 1$ at $\langle 1 \rangle$ which generalizes the original state $a = 1 \wedge b = 1 \wedge y = -1$, yet not as complex as the weakest precondition mentioned above.

Using Polyhedral Library: Given a transition relation $R(\tilde{x}, \tilde{x}')$ on variables $\tilde{x}$ and $\tilde{x}'$, where $\tilde{x}$ represents the program variables before the transition and $\tilde{x}'$ represents the program variables after the transition, and a postcondition $Post(\tilde{x}')$. A weakest precondition is the formula:

$$\begin{aligned} \mathsf{wp}(R(\tilde{x}, \tilde{x}'), Post(\tilde{x}')) &\equiv \forall \tilde{x}' \bullet R(\tilde{x}, \tilde{x}') \rightarrow Post(\tilde{x}') \\ &\equiv \neg(\neg(\forall \tilde{x}' \bullet R(\tilde{x}, \tilde{x}') \rightarrow Post(\tilde{x}'))) \\ &\equiv \neg(\exists \tilde{x}' \bullet R(\tilde{x}, \tilde{x}') \wedge \neg Post(\tilde{x}')) \end{aligned}$$

which now can be estimated using projection (pre-image computation). Here we are only allowed to narrow, but not to widen. We make use of the polyhedral library to ease this computation. The reason is that the polyhedral library allows us to represent a Disjunctive Normal Form (DNF) formula as a union of respective polyhedra. And all the needed operations are closed under this representation (our native CLP($\mathcal{R}$) system does not allow us to represent and manipulate disjunctive

formula directly). The projection to eliminate those variables $\tilde{x}'$ may be an overestimation. However, this is safe as the negation of it will be an underestimation of the weakest precondition. At the end, for efficiency, we only keep a conjunctive formula.

Return to the same example, the weakest precondition for the first path is $(x \geq 0 \lor a > 0)$. However, in getting a conjunctive formula as the interpolant, we decide just to keep $a > 0$. Similarly, what we will keep for the second path is just $b > 0$. As a result, the final interpolant at $\langle 1 \rangle$ will be $(a > 0 \land b > 0)$.

We note that precondition propagation using polyhedral library is more expensive and is only performed when needed. This technique is motivated by the fact that constraint deletion performs badly (much lower chance for reuse) typically when the guard causing infeasibility purely involves symbolic variables. Unfortunately, most of the guards for loops are of this type.

3.7.4 Determining Exactness of the Results

In WCET path analysis, it is important to be able to automatically determine whether the returned bound is *exact*. In our approach, we only lose some path-sensitivity due to path merging at the end of each loop iteration. Obviously, our method produces the *exact* bound for a single-path program. For a loop-free program, our method also computes the *exact* bound. For multi-path programs with loops, our method has an advantage compared to others that we can easily incorporate the techniques in [Thakur and Govindarajan, 2008a] into our algorithm. In short, we initially perform data-flow analysis to determine those control flow merges (called *destructive merges* [Thakur and Govindarajan, 2008a]) which may cause loss in the analysis precision. Then our algorithm can automatically conclude that the returned bound is *exact* if the input program contains no destructive merges.

Benchmark	Description	#LC
bubblesort	Bubble sort program	128
expint	Series expansion for computing an exponential integral function	157
fft1	Fast Fourier Transform using the Cooly-Turkey algorithm	219
fir	Finite impulse response filter (signal processing algorithms)	276
insertsort	Insertion sort program	92
j_complex	Nested loop program with complex flow	64
ns	Search in a multi-dimensional array	535
nsichneu	Automatically generated code containing large amounts of if-statements	2000
ud	LU decomposition algorithm	147
amortized	A program with amortized loop	41
two_shapes	A nested loop where the inner loop is executed only on even-th iteration of outer	20
non_deter	A nested loop having inner loop's counter incremented nondeterministically in each iteration (simpler version of Collatz)	20
tcas	A traffic collision avoidance system, a real life safety critical embedded system	400

Table 3.1: WCET Benchmark Programs

3.8 Experimental Evaluation

We have selected most difficult benchmark programs (with loops and nested loops) from the Mälardalen WCET group [Mälardalen, 2006], namely bubblesort, expint, fft1, fir, insertsort, j_complex , ns, nsichneu (part of it), ud. In addition, tcas, a real life implementation of a safety critical embedded system, is used to illustrate the performance of our method for the case of big loop-free programs. We also introduce 3 academic programs, namely amortized, two_shapes, non_deter to stress more on complicated behaviors of loops. Benchmark descriptions and sizes are briefly summarized in Table 3.1.

We used an Intel Core 2 Duo @ 2.93Ghz with 2GB RAM and built our system upon the CLP($\mathcal{R}$) [Jaffar *et al.*, 1992] and its native constraint solver. Since the benchmark programs are of small and moderate sizes, timeout is set at 300 seconds.

As mentioned earlier, the methods in [Lundqvist and Stenström, 1999; Ermedahl

and Gustafsson, 1997; Gustafsson *et al.*, 2005] are similar to our method with only the iteration abstraction feature (modulo the abstract domain). To have a better comparison, both the performances of our full symbolic simulation method (SS) and its "Iteration Abstraction only" version (IA) are reported in Table 3.2. IA closely mimics the performance of the methods described in [Lundqvist and Stenström, 1999; Ermedahl and Gustafsson, 1997; Gustafsson *et al.*, 2005], especially in term of its complexity wrt. the size parameter. In Table 3.2 we refer to IA as the current state-of-the-art.

In fact, if IA ever returns, its bound will be at least as good as the bound returned by SS. Due to the employment of more accurate abstract domain, IA detects more infeasible paths compared to [Lundqvist and Stenström, 1999; Ermedahl and Gustafsson, 1997; Gustafsson *et al.*, 2005] and therefore its bounds will be tighter than those computed by [Lundqvist and Stenström, 1999; Ermedahl and Gustafsson, 1997; Gustafsson *et al.*, 2005]. For each benchmark, IA also visits less states compared to [Lundqvist and Stenström, 1999; Ermedahl and Gustafsson, 1997; Gustafsson *et al.*, 2005]. There are certain programs that cannot be handled by [Lundqvist and Stenström, 1999; Ermedahl and Gustafsson, 1997; Gustafsson *et al.*, 2005] due to their limited abstract domains, but will be well handled by our IA (see discussion in [Lundqvist and Stenström, 1999]). Of course, it is expensive to maintain a more accurate abstract domain. In particular, we expect that, IA version might take longer running time compared to [Lundqvist and Stenström, 1999; Ermedahl and Gustafsson, 1997; Gustafsson *et al.*, 2005] due to calls to the polyhedral library and the underlying theorem prover for checking feasibility.

Benchmark	Size Parameter (SP)	Actual WCET	Complexity (wrt. SP)	Symbolic Simulation (SS)					State of The Art (IA)		
				States	Time (ms)	WCET	Exact?		States	Time (ms)	WCET
							Manual	Auto			
	n = 25	1648		135	233	1648	Y	N	2873	3087	1648
bubblesort	n = 50	6423	$O(n^2)$	260	701	6423	Y	N	11373	20363	6423
	n = 100	25348		510	2438	25348	Y	N	45248	178268	25348
expint	NA	859	-	519	8247	859	Y	Y	1009	13842	859
	n = 8	181		111	446	181	Y	Y	218	539	181
	n = 16	379		176	927	379	Y	Y	461	1313	379
fft1	n = 32	791	$O(nlogn)$	287	2197	791	Y	Y	970	3764	791
	n = 64	1661		495	6818	1661	Y	Y	2049	14829	1661
fir	NA	760	-	108	387	760	Y	Y	986	5036	760
	n = 25	1120		159	387	1120	Y	N	2861	4847	1120
insertsort	n = 50	4120	$O(n^2)$	309	1504	4120	Y	N	10736	45873	4120
	n = 100	15745		609	7542	15745	Y	N	-	timeout	-
j_complex	NA	133	-	165	491	534	N	N	*	*	*
	n = 5	2655		63	59	2655	Y	Y	5936	4359	2655
ns	n = 10	35555	$O(n^4)$	103	116	35555	Y	Y	86666	104392	35555
	n = 20	522105		183	344	522105	Y	Y	-	timeout	-
nsichneu	NA	281	-	334	15542	281	Y	N	-	timeout	-
ud	NA	819	-	487	1137	819	Y	Y	992	1802	819
	n = 50	394		95	287	394	Y	Y	760	649	394
amortized	n = 100	792	$O(n)$	186	1035	792	Y	Y	1551	2312	792
	n = 200	1590		339	4057	1590	Y	Y	3142	10539	1590
	n = 50	2199		259	797	2199	Y	Y	2874	5068	2199
two_shapes	n = 100	8149	$O(n^2)$	509	3235	8149	Y	Y	10749	42092	8149
	n = 200	31299		1009	19839	31299	Y	Y	-	timeout	-
	n = 25	3904		129	509	3904	Y	Y	8255	17941	3904
non_deter	n = 50	15304	$O(n^2)$	242	1876	15304	Y	Y	-	timeout	-
	n = 100	60604		467	9253	60604	Y	Y	-	timeout	-
tcas	NA	99	-	6020	15925	99	Y	Y	-	timeout	-

Table 3.2: Experiments on WCET Benchmark Programs

For a loop-free program, SS guarantees to produce the *exact bound*[4]. For this kind of program, very importantly, we demonstrate that by using interpolation, we do not necessitate full enumeration of paths. The performances of SS vs. IA for tcas illustrate the point.

As shown in Table 3.2, except for j_complex , SS achieves the exact timing for each of the benchmarks - indicated by column Manual. Some of those, current non-brute-force technique [Prantl *et al.*, 2008] cannot achieve the exact bounds even for certain loops alone. Except for bubblesort, insertsort, nsichneu, j_complex , not only SS produces the exact bounds but also it can *automatically conclude* that it has computed the exact upper bounds - indicated by column Auto - based on the technique elaborated in Section 3.7.4.

The program j_complex , firstly introduced in [Ermedahl and Gustafsson, 1997], was designed in such a way that, as long as path merging is applied at the end of the outer loop body, we overestimate its WCET. As expected, SS does not produce an exact timing for this benchmark, however, the result is still comparable with the method introduced in [Ermedahl and Gustafsson, 1997; Gustafsson *et al.*, 2005] (the number of inner loop iterations is estimated at 66). IA, expected to produce a similar result, however fails because of overflow during a call to the polyhedral library. This is mainly because, in our implementation, we do not support infinite precision computation.

On the other hand, nsichneu is a (multi-path) program with a single loop having a very large body with lots of conditional branches. Its purpose is to test the scalability of an analyzer. Our algorithm does not finish on full nsichneu program (about 4000 LOC) due to the heavy workload on the solver for checking infeasible paths. However, on the attempt to reduce the size of nsichneu, i.e., by reducing the body of the loop by half, our algorithm then not only runs in good time, it also

[4] In theory, this is limited by the power of the theorem prover, since the problem of detecting all infeasible paths is *incomplete*. However, in practice with CLP($\mathcal{R}$), we have encountered no problems regarding this matter.

computes the *exact* bound.

SS finishes in less than 20 seconds for every benchmark program. It significantly outperforms IA in *all* benchmarks. More importantly, for programs of which a size parameter exists and can be easily modified as an input variable, the complexity of our SS is *reduced to linear* (wrt. the size parameter). In most cases, the number of states visited by SS is even smaller than the "Actual WCET", which corresponds to the maximum number of states in a concrete execution of the program. IA, therefore methods in [Lundqvist and Stenström, 1999; Ermedahl and Gustafsson, 1997; Gustafsson *et al.*, 2005], clearly do not possess such properties.

3.9 Other Related Work

WCET path analysis has been the subject of much research, and substantial progress has been made in the area (see [Puschner and Burns, 2000; Wilhelm *et al.*, 2008] for surveys). Implicit Path Enumeration Technique (IPET) [Li and Malik, 1995] and its extensions (e.g., [Engblom and Ermedahl, 2000; Ermedahl *et al.*, 2003; Bygde *et al.*, 2009]) have been widely used due to its efficiency and simplicity. However, pure IPET methods have problems with infeasible paths and flow information stretching across loop-nesting levels. However, complex flow facts can be expressed using user-defined constraints [Engblom and Ermedahl, 2000; Ermedahl *et al.*, 2003], but the complexity of solving the resulting problem is potentially exponential, since the program is completely unrolled and all flow information is lifted to a global level. We note here that, proving the correctness of those constraints is considered as an orthogonal issue.

Considering *infeasible paths* to increase the accuracy of WCET path analysis has attracted a lot of attention in recent years. Nonetheless, all previous works either perform partial detection of infeasible paths (e.g., using conflict sets) [Healy and Whalley, 2002; Suhendra *et al.*, 2006] or suffer from full path enumeration [Park, 1993; Altenbernd, 1996; Stappert *et al.*, 2001].

SATURN [Dillig *et al.*, 2008] and [Reps *et al.*, 1995] are general techniques in program analysis. Ours is related to them since all are summary-based. However, those works are path-insensitive ("where precise means meet-over-all-paths" [Reps *et al.*, 1995]) and cannot be applied to problems which require a high-level of accuracy such as WCET prediction. In contrast, our work attempts path-sensitivity while doing loop summarization. The problem we address is fundamentally different, and much harder.

Our approach poses a commonality with recent CEGAR-based model checking approaches [Ball *et al.*, 2001; Henzinger *et al.*, 2002] in using interpolation concept to eliminate irrelevant facts and optimize the search space. In CEGAR, when coverage happens, a sub-tree can be safely pruned. However, in our case, it only means that the previously analyzed sub-tree (to be exact, its summarization) can be reused under a new context. The difference is due to the fact that here we address a *discovery* and *optimization* problem whereas CEGAR works on decision problem. For instance, as a simpler version of WCET, RCSP (resource-constrained shortest path) by no means can be easily addressed using CEGAR. Indeed, it has been argued before that model checking techniques cannot efficiently deal with WCET analysis [Wilhelm, 2004; Lv *et al.*, 2008].

3.10 Summary

We presented a brute-force path analysis method for inferring and proving tight resource bounds by symbolically simulating loops. The main novelty is first, the use of *interpolation* which allows abstract reasoning which in turn makes the search space manageable; second, the use of *witness paths* to curtail the use of the said abstraction in cases where accuracy is likely to be affected; and finally, the use of *compounded summarizations* on loop iterations in such a way that the state space explored by our symbolic simulation can even be smaller than the number of states in a concrete execution. Using well-known WCET benchmarks, we showed that our

method performs not just well, but often it obtains *exact* results.

We have briefly mentioned earlier that our method naturally supports compositional analysis. However, as shown in Section 3.8, even without it our algorithm still performs well with benchmarks under the real-time system domain. Exploring the usefulness of compositionality property for larger programs is left as our future work.

Chapter 4

Assertions

Wisdom itself is often an abstraction associated not with fact or reality but with the man who asserts it and the manner of its assertion.

John Kenneth Galbraith

Programs use limited physical resources. Thus determining an upper bound on resource usage by a program is often a critical need. Ideally, it should be possible for an experienced programmer to extrapolate from the source code of a well-written program to its *asymptotic* worst-case behavior.

However, "*concrete* worst-case bounds are particularly necessary in the development of embedded systems and hard real-time systems." [Hoffmann *et al.*, 2011]. A designer of a system wants hardware that is *just good enough* to safely execute a given program, in time. As a result, precision is the key requirement in resource analysis of the program. Now embedded programs are often written in favor of performance over simplicity or maintainability. This makes many analytical techniques[1] [Gulwani and Zuleger, 2010; Hoffmann *et al.*, 2011; Esteban and Genaim,

[1] These are more general by the use of *parametric* bounds, and they discover a closed worst-case formula. But they are not generally used for concrete analyses.

2012] less applicable.

Fortunately, there are two important redeeming factors. First, embedded programs are often of small to medium size (from dozens to a few thousand lines of code) and symbolically executing them is guaranteed to terminate. Second, programmers are willing to spend time and effort to help the analyzer in order to achieve more accurate bounds. In many cases, often such manually given *assertions* are essential.

As mention before, we deal with the high level aspects of resource analysis. Architecture modeling, when applicable, is considered as a separate issue and is out of scope of this work. In other words, we only address the *path analysis* problem. Though our path analysis is supposed to work at the level the control flow graphs (CFG), for better comprehension, all the examples we present are at the source code level.

The Need for Path-Sensitivity

```
t = i = 0;
while (i < 10) {
    if (i mod 3 == 0)
        { j *= j; t += 30; }
    else
        { j++; t += 1; }
    i++;
}
```

Figure 4.1: Need for Path Sensitivity

Precise path analysis essentially arises from *path-sensitivity*, and this in turn essentially arises from the ability to disregard *infeasible paths*. But how do we deal with the subsequent explosion in the search space? In fact, due to loops, fully path-sensitive algorithms cannot scale. In practice, abstract reasoning with "path-merging" is used, e.g., [Lundqvist and Stenström, 1999; Ermedahl and Gustafsson, 1997; Gustafsson *et al.*, 2005; Gustafsson *et al.*, 2006].

The example in Fig. 4.1 concerns Worst-Case Execution Time (WCET) analysis,

or simply timing analysis. The special variable t captures the timing. The program iterates through a loop, using the counter i. In the iteration such that `(i mod 3 == 0)`, a multiplication is performed, thus requires 30 cycles to finish. Otherwise, an addition is performed, which takes only 1 cycle to finish. The main challenge is to discover that multiplications are in fact performed three times less often than additions. In general, such discoveries are very hard.

An easy solution is to perform *loop unrolling*. In Fig. 4.1, we start with $(t = 0, i = 0)$. In the first iteration, we detect that the **else** branch is infeasible. At the end of this iteration, we have $(t = 30, i = 1)$ (since j does not affect the control flow, we just ignore information about j). In the second iteration, as $(i = 1)$ we detect that the **then** branch is infeasible; from the other branch, we then obtain $(t = 31, i = 2)$. This process continues until $(i = 10)$, when we exit the loop (having discovered that multiplication is executed exactly 4 times, and the exact WCET of 126).

In short, by unrolling, we precisely capture the value of the counter i which is crucial for determining the infeasible paths across the loop iterations. Though the loop body is executed 10 times, we can capture the fact that the multiplication operation is executed exactly 4 times. Consequently, the bound on the worst case timing is precise (for this case, we indeed derive the exact bound).

Clearly a direct implementation of unrolling cannot scale. In Chapter 3, we developed a *fully automated* symbolic simulation algorithm which is fully path sensitive wrt. loop-free program fragments. For loops, the algorithm performs loop unrolling while employing judicious use of path-merging *only* at the end of each loop iteration, thus useful information is systematically propagated across loop iterations and between different loops. As already pointed out, loop unrolling is almost inevitable in order to capture precisely infeasible path information and complicated loop patterns such as: non-rectangular, amortized, down-sampling, and non-existent of closed form. As discussed, the main contribution of Chapter 3 is the use of sum-

marizations with interpolants in determining reuse so that loop unrolling can be performed *efficiently*.

Importantly, loop unrolling is observed to have *superlinear* behavior for the set of WCET benchmark programs with complicated loops. The key feature that allows scalability of loop unrolling is not just that a single loop iteration is summarized, and then subsequent loop iterations are analyzed using this summarization. Rather, a *sequence* of consecutive loop iterations can be summarized. Potentially, the whole loop is efficiently summarized for later reuse. This is crucial for when it matters most: when loops are nested.

The Need for User Assertions

It is generally accepted in the domain of resource analysis that often programs do not contain enough information for program path analysis. The reason is that programs typically accept inputs from the environment, and behave differently for different inputs. It is just too hard, if not impossible, to automatically extract all such information for the analyzer to exploit.

```
t = 0;
for (i = 0; i < 100; i++) {
    if (A[i] != 0) {
        /* some heavy computation */
        t += 1000;
    } else { t += 1; }
}
```

Figure 4.2: Assertions are Essential

Refer to the example in Fig. 4.2, which is also about timing analysis. Each non-zero element of an array `A[]` triggers a heavy computation. From the program code, we can only infer that the number of such heavy computations performed is bounded by 100. In designing the program, however, the programmers might have additional knowledge regarding the sparsity of the array `A[]`, e.g., no more than 10 percent of

`A[]`'s elements are non-zero. We refer to such user's knowledge as *user assertions*. The ability to make use of such assertions is *crucial* for tightening the worst case bound.

In general, a framework — by accommodating assertions — will allow different path analysis techniques to be combined easily. As path analysis is an extremely hard problem, we do not expect to have a technique that outperforms all the others in all realistic programs. Under a framework which accommodates *assertions*, any *customized* path analysis technique can encode its findings in the (allowed) form of assertions, and simply let the aggregation framework exploit them in yielding tighter worst-case bounds. Two commercial products for timing analysis [aiT, ; Bound-T,] accommodate assertions, giving evidence to their practical importance.

We conclude this subsection by emphasizing that we are not considering the *proof* of assertions in this Chapter, though our algorithm is dependent on the correctness of the assertions. In general, the problem of proving assertions may require a framework that is more general than what is available. This is because on the one hand assertions talk about frequency of execution, on the other, the user knowledge is about the *input*, and often this is in the form of a complex data structure. In any case, we regard the proof of validity of assertions as an *orthogonal* problem.

Path-Sensitivity and Assertions Don't Mix

We have argued that we need both path sensitivity (up to loops) and assertions in order to have precision. In this Chapter, we propose a framework for path analysis in which *precise* context propagation is achieved by unrolling. In addition, we instrument the program with frequency variables, each attached to a basic block. Each frequency variable is initialized to 0 at the beginning of the program and is incremented by 1 whenever the corresponding basic block is executed. Importantly, our framework accommodates the use of assertions on frequency variables so that user information can be explicitly exploited. In other words, the framework not only

attempts *path-sensitivity* via loop unrolling as in Chapter 3, but also makes use of assertions to *block* more paths, i.e., disregard paths which violate the assertions. Ultimately, we can tighten the worst-case bounds.

Now because assertions, when they are used, are typically of high importance, we require our framework to be *faithful to assertions*. That is, all paths violating the given assertions are guaranteed to be excluded from bound calculation process. This requires the framework to be *fully path-sensitive* wrt. the given assertions. However, to make loop unrolling scalable, a form of *greedy* treatment with *path merging* is usually employed. In other words, the analysis of a loop iteration must be finished before we go to the next iteration. Also, the analysis should produce only one *single* continuation context[2]; this context will be used for analysis of subsequent iterations or subsequent code fragments.

It is here that we have a major conflict: unrolling while ensuring that we recognize *blocked* paths that arise because of assertions.

```
c = 0, i = 0, t = 0;
while (i < 9) {
    if (*) {B1: c++; t += 10; }
    else {
        if (i == 1) {B2: t += 5; }
        else {B3: t += 1; }
    }
    i++;
    assert(c <= 4);
}
```

Figure 4.3: Complying with Assertions in Loop Unrolling is Hard

See Fig. 4.3 where the special variable t captures timing, and "*" is a condition which cannot be automatically reasoned about, e.g., a call to an external function `prime(i)`. We also instrument the program with the frequency variable c which is incremented each time **B1** is executed. The assertion `assert(c <= 4)` constrains that **B1** can be executed at most 4 times.

[2] We can generalize this to a fixed number of continuation contexts.

We now exemplify loop unrolling. In the first iteration, we consider the first **then** branch, notice t is incremented by 10 so that we get $(t = 10, c = 1, i = 1)$ at the end. Considering however the **else** branch, we then detect that the **then** branch of nested **if-statement** is infeasible, thus we finally get $(t = 1, c = 0, i = 1)$. Performing path-merging to abstract these two formulas, we conclude that this iteration can consume up to 10 cycles $(t = 10)$ and we continue to the next iteration with the context $(t = 10, c = 0 \vee c = 1, i = 1)$[3]. Note that by path-merging, we no longer have the precise information about c, i.e., we say this merge is *destructive* [Thakur and Govindarajan, 2008b].

Now there are two options in using the assertion to block invalid paths. They follow the *must* and *may* semantics, respectively.

First, our strategy is to block path with context which *must* violate the assertion. However, in our example, due to the destructive merge at the end of each iteration, no path will be blocked (by the provided assertion). Consequently, we end up having the worst case timing is 90 cycles: each iteration consumes 10 cycles. Importantly, the provided assertion cannot be used to tighten the bound.

Second, an alternative is to block path with context which *may* violate the assertion. In the first four iterations, the execution of block **B1** is possible. From the fifth iteration onwards, this strategy forbids those paths executing **B1**. As such, from the fifth to the tenth iteration, the only feasible path is by following the **else** branch of **if (i == 1)** statement. This leads to the timing of 45 at the end.

At first glance, the second strategy seems to be able to make use of the provided assertion in order to block paths and tighten the bound. However, such analysis is *unsound*. A counter-example can be achieved by replacing `if (*)` with `if (prime(i))`, where `prime` is a function which returns *true* if the input is actually a *prime* number and *false* otherwise. This counter-example has the timing of $(1 + 5 + 10 + 10 + 1 + 10 + 1 + 10 + 1 = 49)$.

[3] To be practical, one must employ some abstract domain for such merge. In our implementation, we use the polyhedral domain.

In summary, to be sound while compliant with assertions, our framework is required to be *fully path-sensitive* wrt. the variables used in the assertions. The greedy treatment of loops currently prevents us from being so. In other words, the challenge is how to address what is in general an intractable *combinatorial problem*.

Main Contribution

This Chapter proposes the *first* analysis framework that is path-sensitive while, at the same time, natively supports user assertions. The famous Implicit Path Enumeration Technique (IPET) [Li and Malik, 1995] naturally supports assertions; it is, however, path-insensitive. To obtain precise analysis, the users need to manually provide information regarding the loop bounds and infeasible paths. On the other hand, Chapter 3 has shown that path-sensitive analysis with loop unrolling can be performed efficiently. However, as we have argued, supporting assertions in a loop unrolling framework is *non-trivial*.

We address the challenge by presenting an algorithm where the treatment of each loop is separated in two phases. Scalability, in both phases, is achieved using the concept of summarization with interpolant. We note here that, as programs usually contain more than one loop and also nested loops, our two phases are, in general, *intertwined*.

The first phase performs a symbolic execution where loops are unrolled efficiently. In order to control the explosion of possible paths, a merge of contexts is done at the end of every loop iteration. While this is an abstraction, it in general produces different contexts for different loop iterations. Thus this is the basis for being path-sensitive up to loops while capturing the non-uniform behavior of different loop iterations.

Different from Chapter 3, the main objective of our loop unrolling (in the first phase) is to simplify the tree by eliminating two kinds of paths:

- those that are *infeasible* (detected from path-sensitivity), and

- those that are *dominated*. In more detail, for each collection of paths that modify the variables used in assertions *in the same way*, only one path (in that collection) whose resource usage *dominates* the rest will be kept in the summarization.

What results from the first phase is a greatly simplified execution tree. The explored tree is then compactly represented as a transition system, which also is a directed acyclic graph (DAG), each edge is labelled with a resource usage and how the assertion variables are modified.

In the second phase, we are no longer concerned with path-sensitivity of the original program, but instead are concerned only about assertions. More specifically, from the produced transition system, we need to disregard all paths *violating* the assertions. The problem is thus an instance of the classic Resource Constrained Shortest Path problem [Joksch, 1966]. While this problem is NP-hard, however, it has been demonstrated in [Jaffar *et al.*, 2008] that, in general, the use of summarization with interpolant can be very effective.

Finally, we give evidence of this with some practical benchmarks.

4.1 Related Work

The State-of-the-Art: Implicit Path Enumeration

Implicit Path Enumeration Technique (IPET) [Li and Malik, 1995] is the state-of-the-art for path analysis in the domain of Worst Case Execution Time (WCET) analysis. IPET formulates the path analysis problem as an optimization problem over a set of frequency variables each associated with a basic block. More precisely, it starts with the control flow graph (CFG) of a program, where each node is a basic block. Program flow is then modeled as an assignment of values to *execution count variables*, each c_{entity} of them associated with a basic block of the program. The values reflect the total number of executions of each node for an execution of the

program.

Each basic block entity with a count variable (c_{entity}) also has a timing variable (t_{entity}) giving the timing contribution of that part of the program to the total execution (for each time it is executed). Generally, t_{entity} is derived by some low-level analysis tool in which micro-architecture is modeled properly. This is an orthogonal issue and is out of the scope of this work.

The possible program flows given by the structure of the program are modeled by using structural constraints over the frequency variables. Structural constraints can be automatically constructed using Kirchhoff's law. Because these constraints are quite simple, IPET has to rely on additional constraints, i.e., *user assertions*, to differentiate feasible from infeasible program flows. Some constraints are *mandatory*, like upper bounds on loops; while others will help tighten the final WCET estimate, like information on infeasible paths throughout the program. Some examples on discovering complex program flows and then feed them into IPET framework are [Engblom and Ermedahl, 2000; Ermedahl *et al.*, 2003].

In the end, the WCET estimate is generated by maximizing, subject to flow constraints, the sum of the products of the execution counts and execution times:

$$\mathsf{WCET} = maximize(\sum_{\forall entity} c_{entity} \cdot t_{entity})$$

This optimization problem is handled using Integer Linear Programming (ILP) technique. Note that IPET does not find the worst-case execution path but just gives the worst-case count on each node. There is no information about the precise execution order.

In comparison with our work in this Chapter, *aside from accuracy*, we have two additional important advantages:

- IPET cannot be extended to work for *non-cumulative* resource analysis such as

memory high watermark analysis. The reason is that such analysis, even for a single path, depends on the order in which statements are executed; while in IPET formulation, such information is abstracted away. We elaborate on this when discussing Fig. 4.8 below.

- IPET supports only *global assertions*, whereas we support both global and *local assertions*. IPET relies on the intuition that it is relatively easy for programmers to provide assertions in order to disregard certain paths from bound calculation, probably because they are the developers. We *partly* agree with this. There is flow information, which can be hard to discover, but the programmers can be well aware of it, a calculation framework should take into account such information to tighten the bounds. Nevertheless, we cannot expect the programmers to know *everything* about the program. For example, it is unreasonable to expect the programmer to state about a path which is infeasible due to a combination of guards scattered throughout the whole program. Such global knowledge is hard to deduce and could well be as hard as the original path analysis problem. In short, it is reasonable to only assume that the programmers know about *some local behavior* of a code fragment, not everything about the global behavior of all the program paths. We elaborate on this when discussing Fig. 4.7 below.

Symbolic Simulation with Loop Unrolling

In the domain of resource analysis, precision is of paramount importance. Originally, *precision* was addressed by symbolic execution with loop unrolling [Lundqvist and Stenström, 1999; Ermedahl and Gustafsson, 1997; Gustafsson *et al.*, 2005; Gustafsson *et al.*, 2006]. A loop-unrolling approach which symbolically executes the program over all permitted inputs is clearly the most accurate. The obvious challenge is that this is generally not scalable. Thus *path-merging*, a form of abstract interpretation [Cousot and Cousot, 1977], is introduced to remedy this fact.

It, in one hand, improves scalability; on the other hand, it seriously hampers the precision criterion.

The most recent related work is in Chapter 3, also presented in [Chu and Jaffar, 2011], which is a basis of the work in this Chapter. Its main technique to reduce both the depth and the breadth of the symbolic execution tree is by making use of *compounded summarization*. This gives rise to the *superlinear* behavior of program with nested loops. That is, the number of states visited in the symbolic execution tree can be asymptotically smaller than the number of states in a concrete run. As a result, path-merging is still performed, but now sparsely only at the end of each loop iteration.

Path-merging has the effect of combining a disjunction of formulas, each representing a context obtained from the path, into a single conjunction that is manageable. However, because of this abstraction, *exact reasoning* is forsook. The distinction of [Chu and Jaffar, 2011] is that the abstraction is designed to preserve information that is common through *each* loop iteration, in contrast with the standard "loop invariant" approach where the abstraction sought is for information that is common through *all* loop iterations.

Parametric Bounds

Static resource analysis concerns with either *parametric* or *concrete* bounds. Parametric methods, e.g., [Gulwani and Zuleger, 2010; Hoffmann *et al.*, 2011; Esteban and Genaim, 2012], study the loops, recursions, and data structures, in order to come up with a closed, easy to understand worst-case formula. These methods are ideal to algorithmically explain the worst-case complexity of resource usage.

But these methods, in general, do not give precise enough bounds (they concern with asymptotical precision only) and are applicable only to a small class of programs. For instance, most systems are restricted to linear constraints. Furthermore, they usually focus on an individual loop or loop nest, rather than capture

the combining effects of loops and conditional statements throughout the program. Consequently, such techniques alone are mainly used for proving program termination.

We believe, however, that the advance in producing parametric bounds for complicated loops can indeed support *concrete* resource analysis, provided that an aggregation framework can make use of assertions.

4.2 Motivating Examples

EXAMPLE 4.1 : See Fig. 4.1. We have shown that loop unrolling produces *exact* timing analysis for this example. Here we show how IPET could exploit user assertions in order to achieve the same. Before proceeding, we mention that this example was highlighted in [Li and Malik, 1995] to demonstrate the use of assertions in the IPET framework.

```
t = i = c = c1 = c2 = 0;
while (i < 10) {
    c++;
    if (i mod 3 == 0)
        { c1++; j *= j; t += 30; }
    else
        { c2++; j++; t += 1; }
    i++;
}
```

Figure 4.4: Assertions in IPET

Now see Fig. 4.4 where frequency variables c, c_1, and c_2 are instrumented. The structural constraint prescribed by the IPET method is $c = c_1 + c_2$. (In general, structural constraints are easily extracted from the CFG.) The objective function to be maximized in the IPET formulation for this example is $(c_1 * 30 + c_2 * 1)$. In order to be exact, one could use the assertion $c \leq 10$ and $c_1 \leq 4$. Note that bounding c (i.e., the assertion $c \leq 10$) is *mandatory* because a bound on the objective function

depends on this. On the other hand, the assertion $c_1 \leq 4$ is *optional*. Computing the optimal now will produce the exact timing.

This example also exemplifies the fact that IPET is perfectly suited to assertions simply because one just needs to conjoin the assertions to the structural constraints before performing the optimization.

EXAMPLE 4.2 : It is certainly not the case that assertions alone are sufficient in general. Let us now slightly modify Ex. 4.1. We replace `j *= j` by `i *= i`. The new program is shown in Fig. 4.5. Following the unrolling technique, *exact* bound is still achieved. The reason is that context propagation is performed precisely. However, it is now *hard* for the user of IPET framework to come up with assertions on frequency variables in order to achieve some good bound. This scenario also poses a big challenge for many analytical methods [Gulwani and Zuleger, 2010; Hoffmann *et al.*, 2011; Esteban and Genaim, 2012], due to the *non-linear* operation on i, i.e., the statement `i *= i;`.

```
t = i = c = c1 = c2 = 0;
while (i < 10) {
    c++;
    if (i mod 3 == 0)
        { c1++; i *= i; t += 30; }
    else
        { c2++; j++; t += 1; }
    i++;
}
```

Figure 4.5: Assertions Alone Are Not Enough

With this example, our purpose is *not to* refute the usefulness of assertions. Instead, we want to further emphasize the ability of an analyzer in propagating flow information precisely and automatically, i.e., the ability of being path-sensitive.

EXAMPLE 4.3 : Let us refer back to the example in Fig. 4.2. Now our focus is on how frequency variables and assertions should be instrumented.

First, note that our frequency variables are similar to frequency variables in IPET

framework. One frequency variable is attached to a *distinct* basic block. Each is initialized to 0 at the beginning of the program, and incremented by 1 whenever the corresponding basic block is executed. An important difference from IPET is that our frequency variables can be *reset*. This gives rise to the use of *local assertions*, and we elaborate on this below.

Our assertions are predicates over frequency variables and for simplicity, will only be provided at the end of some loop body or at the end of the program (otherwise a preprocessing phase is needed).

In Fig. 4.6, the assertion captures the fact that the input array A is a *sparse* one: no more 10 percent of its elements are non-zero.

```
t = c = c1 = 0;
for (i = 0; i < 100; i++) {
    c++;
    if (A[i] != 0) {
        c1++;
        t += 1000;
    } else { t += 1; }
}
assert(c1 <= c / 10);
```

Figure 4.6: Assertions Are Essential

EXAMPLE 4.4 : Consider the following "bubblesort" example in Fig. 4.7 where we have placed a frequency variable c. An integer array `a[]` of size $N > 0$ is the input. Every element of `a[]` belongs to the integer interval $[min, max]$. Now, assume that we know that there are M elements which are equal to max. Consider: how many times are a pair of elements swapped? We believe this is currently beyond any systematic approach.

However, it is relatively easy to derive the assertion shown in the inner loop, representing *local reasoning*: each swap involves an element which is not equal to max on the right, therefore after the swap such element would not be visited again

in the subsequent iterations of the inner loop. Consequently, for each invocation of the inner loop, the number of swaps is no more than the number of elements which are not equal to max.

Note that the frequency variable c is reset right before each invocation of the inner loop. If this were not done, then, to find an alternative assertion to `(c <= N-M)` may not be feasible. That is, a global assertion (in this case, to assert that the total number of increments to c) is in general *much harder* to discover than a local one.

```
for (i = N-1; i >= 1; i--) {
    c = 0;
    for (j = 0; j <= i-1; j++)
        if (a[j] > a[j+1]) {
            c++;
            tp = a[j]; a[j] = a[j+1]; a[j+1] = tp;
            t += 100;
        } else { t += 1; }
        assert(c <= N-M);
    }
}
```

Figure 4.7: Local Assertions

EXAMPLE 4.5 : Refer to the program in Fig. 4.8 which concerns memory high watermark analysis. This is an example of *non-cumulative* resource usage. The case for using loop unrolling is particularly clear for such analysis.

The special variable m captures the amount of memory has been consumed. In memory high watermark analysis, the order in which memory is allocated and deallocated plays a crucial role in determining a tight bound. Note that the IPET formulation abstracts away the ordering within a program path, thus this approach is in fact not appropriate.

The function `parity` is external, and n is an input variable, so we cannot reason about the condition `parity(n)`. Performing loop unrolling with path merging at the end of each iteration, we get the worst case bound of 1010. This corresponds to

```
⟨1⟩ c1 = c2 = 0;
⟨2⟩ m = 0;                              /* initially */
⟨3⟩ m = m + 10;                  /* malloc statement */
⟨4⟩ for (i = 0; i < 100; i++) {
⟨5⟩     if (parity(n)) {          /* function call */
⟨6⟩         c1++;
⟨7⟩         m = m + 10;          /* malloc statement*/
        } else {
⟨8⟩         c2++;
⟨9⟩         m = m - 10;            /* free statement */
        }
⟨10⟩    n++;
⟨11⟩    assert(|c1 - c2| <= 1);
    }
⟨12⟩
```

Figure 4.8: Memory High Watermark Analysis

the spurious path that executes the **then** body of the **if-statement** in all iterations. Note that this bound, though extremely imprecise, is indeed *safe*. However, as a programmer, the user/certifier might know that the loop will execute alternately the **then** and the **else** bodies (though he does not know which body the first iteration will execute, due to the unknown value of input `n`). Therefore, the programmer can provide the assertion `assert(|c1 - c2| <= 1)` to guide the reasoning to an *alternating-like* behavior of the loop. With this assertion, the returned worst case high watermark would be as low as 20, instead of 1010.

The given program fragment in general can be put inside an outer loop. The provided assertion then is correct only for each invocation of the loop at hand. This again demonstrates the power of our *local* assertions.

4.3 Preliminaries

Similar to Chapter 3, our programs now include a special variable: the resource variable `r`. For simplicity, we start with cumulative resource usage such as time, power. Thus the purpose of our path analysis is to compute a *sound* and *accurate*

bound for r in the end, across all feasible paths of the program. We note here that later in Section 4.5.1, we extend our work to support analysis of non-cumulative resource such as memory, bandwidth.

Also recall our frequency variables are similar to frequency variables in IPET framework. One frequency variable is attached to a *distinct* basic block. They are initialized to 0 at the beginning of the program, and incremented by 1 whenever the corresponding basic block is executed. An important difference from IPET is that our frequency variables can be *reset*, which gives rise to the use of *local assertions*.

Our assertions are predicates over frequency variables and for simplicity, can be provided at the end of some loop body, or at the end of the program. A normal assertion can be easily transformed to an equivalent one which conforms to this requirement.

Definition 20 (Assertion)**.** *An* assertion *is a tuple $\langle \ell, \phi(\tilde{c}) \rangle$, where ℓ is a program point and $\phi(\tilde{c})$ is a set of constraints over the frequency variables $\tilde{c}$.* □

Definition 21 (Blocked State)**.** *Given a feasible state $s \equiv \langle \ell, [\![s]\!] \rangle$ and an assertion $\mathcal{A} = \langle \ell, \phi(\tilde{c}) \rangle$, we say state s is* blocked *by assertion $\mathcal{A}$ if $[\![s]\!] \wedge \phi(\tilde{c})$ is unsatisfiable.* □

Definition 22 (Summarization of a Subtree)**.** *Given two program points ℓ_1 and ℓ_2 such that ℓ_2 post-dominates ℓ_1 and assume we analyze all the paths from entry point ℓ_1 to exit point ℓ_2 wrt. an incoming context $[\![s]\!]$. The summarization of this subtree is defined as the tuple $[\ell_1, \ell_2, \Gamma, \Delta, \overline{\Psi}]$, where Γ is the set of* representative paths*,* abstract transformer *Δ is a binary input-output relation between variables at ℓ_1 and ℓ_2, and* interpolant *$\overline{\Psi}$ is the condition under which this summarization can be safely reused.* □

Computations of Δ and $\overline{\Psi}$ are the same as in Chapter 3. Now we explain the new element, the set of representative paths Γ.

All the feasible paths of the subtree at hand are divided into a number of classes, each modifying the frequency variables in a distinct way. We are interested in only

frequency variables which are *live* at the exit point ℓ_2 and will be used later in some assertion. We call those frequency variables the *relevant* ones.

Now for each class, only the *dominating* path — the one with highest resource consumption — will be kept in Γ. Specifically, each representative path $\gamma \in \Gamma$ is of the form $\langle \mathsf{r}_0, \delta_0(\tilde{c}, \tilde{c}') \rangle$, where r_0 is the amount of resource consumed in that path and $\delta_0(\tilde{c}, \tilde{c}')$ captures how frequency variables are modified in that path.

The fact that two representative paths $\gamma_1 = \langle \mathsf{r}_1, \delta_1(\tilde{c}, \tilde{c}') \rangle$ and $\gamma_2 = \langle \mathsf{r}_2, \delta_2(\tilde{c}, \tilde{c}') \rangle$ modify the set of (relevant) frequency variables in the same way is denoted by $\delta_1(\tilde{c}, \tilde{c}') \overset{\mathcal{A}}{\equiv} \delta_2(\tilde{c}, \tilde{c}')$.

For convenience, we repeat the definition for a summarization of a program point here.

Definition 23 (Summarization of a Program Point)**.** *A summarization of a program point ℓ is the summarization of all paths from ℓ to ℓ' (wrt. the same context), where ℓ' is the nearest program point that post-dominates ℓ s.t. ℓ' is of the same nesting level as ℓ and either is (1) an ending point of the program, or (2) an ending point of some loop body.*

As ℓ' can always be deduced from ℓ, in the summarization of program ℓ, we usually omit the component about ℓ'.

4.4 The Algorithm: Overview of the Two Phases

Phase 1: The first phase uses a greedy strategy in the unrolling of loops, Chapter 3. This unrolling explores a conceptually symbolic execution tree, which is of enormous size. One main purpose of this phase is to precisely propagate the context across loop iterations, and therefore disregard as many infeasible paths as possible from consideration in the second phase.

For each iteration, for all feasible paths discovered, we divide them into a number of classes. Paths belong to a class modify the frequency variables in the same way.

Paths from different classes modify the frequency variables in different ways. As an optimization, we are only interested in frequency variables which will later be used in some assertion. For each class, only the path with highest resource consumption is kept, we say that path is the *dominating* path of the corresponding class. Thus another purpose of the first phase is to disregard dominated paths from consideration in the second phase.

From the dominating paths discovered, we now represent compactly this iteration as a set of transitions. This representation is manageable because we can restrict attention only to the frequency variables used later in some assertion. We continue this process iteration by iteration. For two different iterations, the infeasible paths detected in each iteration can be quite different. As a result, their representations will be different too. At the end of phase 1, we represent the unrolled loop in the form of a transition system in order to avoid an upfront consideration of the search space for the whole loop, which can potentially still be exponential.

```
⟨1⟩    c = 0, i = 0, t = 0;
⟨2⟩    while (i < 9) {
⟨3⟩        if (*) {
⟨4⟩            if (*) {
⟨5⟩                c++; t += 10;
               } else {
⟨6⟩                t += 1;
               }
           } else {
⟨7⟩            if (i == 1) {
⟨8⟩                t += 5;
               } else {
⟨9⟩                t += 1;
               }
           }
⟨10⟩       i++;
⟨11⟩       assert(c <= 4);
⟨12⟩   }
⟨13⟩
```

Figure 4.9: Complying with Assertions in Loop Unrolling

EXAMPLE 4.6 : Consider the program fragment in Fig. 4.9, which is slightly modified from the example shown in Fig. 4.3 (and now with instrumented program points). Note that at phase 1, we ignore the assertion at program point ⟨11⟩, but pay attention only to its frequency variable c.

We enter the first iteration of the loop. Inside the loop body, we follow the first feasible path (⟨3⟩ ⟨4⟩ ⟨5⟩ ⟨10⟩ ⟨12⟩). The value of i at ⟨12⟩ is 1. When backtracking, a summarization of program point ⟨10⟩ is computed as:

$$[\langle 10\rangle, \{\langle 0, \mathsf{c} := \mathsf{c}\rangle\}, i' = i + 1, \mathit{true}]$$

In other words, the subtree from ⟨10⟩ to ⟨12⟩ is summarized by: (1) a representative path which does not consume any resource and does not modify the assertion variable c either; (2) an abstract transformer which says that the output value of i is equal to the input value of i plus 1, (3) an interpolant *true* which means that any state at program point ⟨10⟩ can safely reuse this summarization. Similarly, we derive a summarization for program point ⟨5⟩ as:

$$[\langle 5\rangle, \{\langle 10, \mathsf{c} := \mathsf{c}{+}1\rangle\}, i' = i + 1, \mathit{true}]$$

The main difference here is that the representative path from ⟨5⟩ to ⟨12⟩ consumes 10 units of time and increments the assertion variable c by 1. From ⟨4⟩, we now follow the **else** branch of the second **if-statement** to reach ⟨6⟩ and then ⟨10⟩. At ⟨10⟩ we reuse the computed summarization for ⟨10⟩. The abstract transformer $i' = i + 1$ is used to produce the continuation context for i at ⟨12⟩ ($i = 1$). We then backtrack and a summarization for ⟨6⟩ is computed as:

$$[\langle 6\rangle, \{\langle 1, \mathsf{c} := \mathsf{c}{+}1\rangle\}, i' = i + 1, \mathit{true}]$$

Thus the combined summarization for ⟨4⟩ (from ⟨5⟩ and ⟨6⟩) is:

$$[\langle 4\rangle, \{\langle 10, \mathsf{c} := \mathsf{c}{+}1\rangle, \langle 1, \mathsf{c} := \mathsf{c}\rangle\}, i' = i + 1, \mathit{true}]$$

Note that this summarization of ⟨4⟩ contains two representative paths, since there are two distinct ways in modifying the assertion variable c. From ⟨3⟩ we now follow the **else** branch of the first **if-statement**. Since i is currently 0, going from ⟨7⟩ to ⟨8⟩ (the **then** branch of the third **if-statement**) is infeasible. This fact is summarized

as:

$[\langle 8\rangle, \emptyset, \mathit{false}, \mathit{false}]$

Following the **else** branch we reach $\langle 9\rangle$ and then $\langle 10\rangle$. At $\langle 10\rangle$ we reuse and backtrack. The summarization of $\langle 9\rangle$ is computed as:

$[\langle 9\rangle, \{\langle 1, \mathsf{c} := \mathsf{c}\rangle\}, i' = i + 1, \mathit{true}]$

Thus the combined summarization for $\langle 7\rangle$ (from $\langle 8\rangle$ and $\langle 9\rangle$) is:

$[\langle 7\rangle, \{\langle 1, \mathsf{c} := \mathsf{c}\rangle\}, i' = i + 1, i \neq 1]$

Note how the infeasible paths from $\langle 7\rangle$ to $\langle 8\rangle$ affects the interpolant for the summarization at $\langle 7\rangle$. Now we need to combine the summarizations of $\langle 4\rangle$ and $\langle 7\rangle$ to get a summarization for $\langle 3\rangle$. We can see that the second representative path in the summarization of $\langle 4\rangle$ and the only representative path in the summarization of $\langle 7\rangle$ both do not modify the frequency variable c and consume 1 unit of time. In other words, each of them dominates the other. Consequently, we only keep one of them in the summarization of $\langle 3\rangle$. The interpolants for $\langle 4\rangle$ and $\langle 7\rangle$ are propagated back (we use precondition computation) and conjoined to give the interpolant for $\langle 3\rangle$. The summarization of $\langle 3\rangle$ wrt. the context of the first iteration of the loop is then:

$[\langle 3\rangle, \{\langle 10, \mathsf{c} := \mathsf{c}{+}1\rangle, \langle 1, \mathsf{c} := \mathsf{c}\rangle\}, i' = i + 1, i \neq 1]$

For the first iteration, we add into our new transition system (we omit $\mathsf{c} := \mathsf{c}$ in the second transition):

$\langle\langle\langle 2\rangle\text{-}0\rangle, \mathsf{c} := \mathsf{c}{+}1 \wedge \mathsf{t} := \mathsf{t}{+}10, \langle\langle 2\rangle\text{-}1\rangle\rangle$

$\langle\langle\langle 2\rangle\text{-}0\rangle, \mathsf{t} := \mathsf{t}{+}1, \langle\langle 2\rangle\text{-}1\rangle\rangle$

The second iteration begins with the context $i = 1$. At program point $\langle 3\rangle$, as the current context does not imply the interpolant $i \neq 1$ of the existing summarization for $\langle 3\rangle$, reuse does not happen. Follow the **then** branch, we reach program point $\langle 4\rangle$ and we can reuse the existing summarization of $\langle 4\rangle$, also produce a continuation context $i = 2$ using the abstract transformer $i' = i + 1$. We then visit program point $\langle 7\rangle$ where we cannot reuse previous analysis. Different from the first iteration, going

from ⟨7⟩ to ⟨8⟩ is now feasible while going from ⟨7⟩ to ⟨9⟩ is infeasible. As a result, a new summarization for ⟨7⟩ is computed as:

$[\langle 7\rangle, \{\langle 5, \mathsf{c} := \mathsf{c}\}\rangle, i' = i+1, i = 1]$

Subsequently, a summarization of ⟨3⟩ wrt. the context of the second iteration is computed as:

$[\langle 3\rangle, \{\langle 10, \mathsf{c} := \mathsf{c+1}\rangle, \langle 5, \mathsf{c} := \mathsf{c}\}\rangle, i' = i+1, i = 1]$

For the second iteration, we add in the following transitions:

$\langle\langle\langle 2\rangle\text{-}1\rangle, \mathsf{c} := \mathsf{c+1} \land \mathsf{t} := \mathsf{t+10}, \langle\langle 2\rangle\text{-}2\rangle\rangle$

$\langle\langle\langle 2\rangle\text{-}1\rangle, \mathsf{t} := \mathsf{t+5}, \langle\langle 2\rangle\text{-}2\rangle\rangle$

Analyses of subsequent iterations reuse the analysis of the first iteration (since the contexts imply the interpolant $i \neq 1$). The following transitions — from iteration j to iteration $j+1$, where $j = 2..8$ — will be added into the new transition system. Note that we also add the last transition which corresponds to the loop exit.

$\langle\langle\langle 2\rangle\text{-}2\rangle, \mathsf{c} := \mathsf{c+1} \land \mathsf{t} := \mathsf{t+10}, \langle\langle 2\rangle\text{-}3\rangle\rangle$

$\langle\langle\langle 2\rangle\text{-}2\rangle, \mathsf{t} := \mathsf{t+1}, \langle\langle 2\rangle\text{-}3\rangle\rangle$

$\ldots$

$\langle\langle\langle 2\rangle\text{-}8\rangle, \mathsf{c} := \mathsf{c+1} \land \mathsf{t} := \mathsf{t+10}, \langle\langle 2\rangle\text{-}9\rangle\rangle$

$\langle\langle\langle 2\rangle\text{-}8\rangle, \mathsf{t} := \mathsf{t+1}, \langle\langle 2\rangle\text{-}9\rangle\rangle$

$\langle\langle\langle 2\rangle\text{-}9\rangle, , \langle 13\rangle\rangle$

We note here that the representation of the second iteration is different from the representations of all other iterations. This arises from the fact that the iterations of the loop do not behave uniformly. This can only be captured by loop unrolling while being path-sensitive.

Before proceeding, we first comment that the simplification that phase 1 performs is often very significant. This is essentially because each iteration can exploit the path-sensitivity that survives through the merges of previous iterations. In general, this leads to an exponential decline in the total number of paths in the resulting

transition system.

Phase 2: In the second phase, we attack the remaining problem, to determine the longest path in this new transition system, also using the concept of summarization with interpolant. The key point to note is that only at this second phase, assertions are taken into consideration to block paths. Consequently, paths violating the assertions will be considered as infeasible, i.e., they are disregarded from bound calculation.

Let us continue with Ex. 4.6. Consider the transition system from phase 1. We now need to solve the longest path problem using the initial context $(c = 0, t = 0)$. The original assertion will be checked at every program point $\langle\langle 2\rangle\text{-j}\rangle$ for $j = 1..9$. To be faithful to the given assertion, we must disregard all paths which increment c more than 4 times from consideration.

In phase 2, our analysis using summarization with interpolant is now performed on a new transition system which contains no loops. Given the transition system produced by phase 1, a naive method would require to explore 894 states from 2^9 paths. By employing summarizations with interpolants the number of explored states is reduced to 56, of which 24 states are reuse states. The effectiveness of summarization with interpolant for such problem instances has already been demonstrated in [Jaffar *et al.*, 2008].

Instead of walking through (again) the application of summarization with interpolant, we offer some insights as to why phase 2 is often tractable (though the expanded tree is still quite deep). See Fig. 4.10, where circles denote "reuse" states and require no further expansion. Note that although the DAG contains exponentially many paths, there are only a few contexts of interest, namely $c = \alpha$ where $0 \leq \alpha \leq 9$. Therefore, for each node in the DAG there needs only 10 considerations of paths starting from the node. Thus a straightforward dynamic programming approach would suffice. However, it is important to note that in general (for example, more than one frequency variable), the number of different contexts of each node is

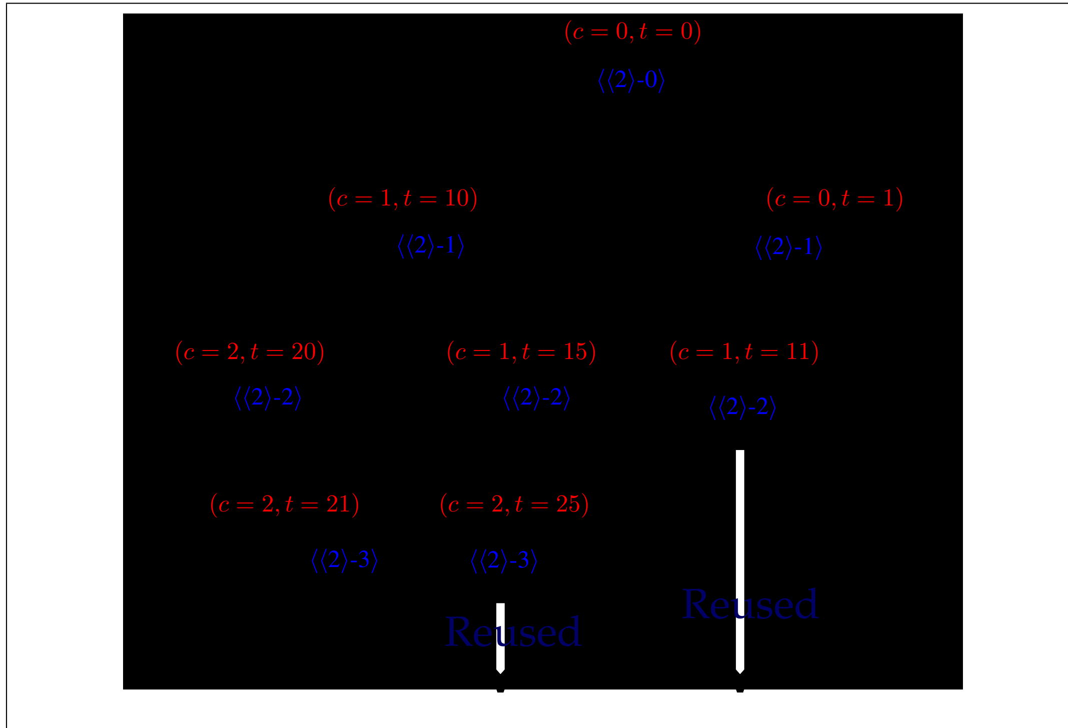


Figure 4.10: Reduced Search Space in Phase 2

exponential. The algorithm we use has the special advantage of using interpolation so that the dynamic programming effect can be enjoyed by considering not just the context, but some sophisticated *generalization* of the context. Essentially, two contexts can in fact be considered equal if they exhibit the same infeasible/blocked paths.

4.5 The Algorithm: Technical Description

We start our analyzer by calling function Analyze with the initial state s_0 and the input transition system $\mathcal{P}$. We start phase 1 (line 1) of the analysis and use the results to build a new transition system $\mathcal{G}$ (line 3). The function Build takes as inputs a set of representative paths Γ_0, the transition system $\mathcal{P}$ of the original program, and an initial transition system $\mathcal{G}$. It adds into $\mathcal{G}$ a number of transitions, which correspond the representative paths in Γ_0. That new transition system is

function Analyze($s_0, \mathcal{P}$)
⟨1⟩ $[\cdot, \Gamma_0, \cdot, \cdot]$:= Summarize($s_0, \mathcal{P}, 1$)
⟨2⟩ $\mathcal{G} := \emptyset$
⟨3⟩ $\mathcal{G}$:= Build($\Gamma_0, \mathcal{P}, \mathcal{G}$)
⟨4⟩ $[\cdot, \Gamma, \cdot, \cdot]$:= Summarize($s_0, \mathcal{G}, 2$)
⟨5⟩ **return** FindMax(Γ)

function Summarize($s, \mathcal{P}$, phase)
Let s be $\langle \ell, [\![s]\!] \rangle$
⟨6⟩ **if** ($[\![s]\!] \equiv \mathit{false}$) **return** $[\ell, \emptyset, \mathit{false}, \mathit{false}]$
⟨7⟩ **if** (outgoing($\ell, \mathcal{P}$) $\equiv \emptyset$) **return** $[\ell, \{\langle 0, \mathsf{Id}(\tilde{c})\rangle\}, \mathsf{Id}(\mathit{Vars}), \mathit{true}]$
⟨8⟩ **if** (loop_end($\ell, \mathcal{P}$)) **return** $[\ell, \{\langle 0, \mathsf{Id}(\tilde{c})\rangle\}, \mathsf{Id}(\mathit{Vars}), \mathit{true}]$
⟨9⟩ S := memoed(s)
⟨10⟩ **if** (S $\not\equiv \mathit{false}$) **return** S
⟨11⟩ **if** (phase $\equiv 2$) /* Consider assertions at phase 2 */
⟨12⟩ **if** ($\exists\, \mathcal{A} \equiv \langle \ell, \phi \rangle$ *and* $[\![s]\!] \wedge \phi \equiv \mathit{false}$) **return** $[\ell, \emptyset, \mathit{false}, \neg\phi]$
⟨13⟩ **if** (phase $\equiv 1 \wedge$ loop_entry($\ell, \mathcal{P}$))
⟨14⟩ $s_i := s$
⟨15⟩ $\mathcal{G} := \emptyset$
⟨16⟩ $[\cdot, \Gamma_1, \Delta_1, \cdot]$:= TransStep($s_i, \mathcal{P}$, entry($\ell, \mathcal{P}$), 1)
⟨17⟩ **while** ($\Gamma_1 \not\equiv \emptyset$)
⟨18⟩ $\mathcal{G}$:= Build($\Gamma_1, \mathcal{P}, \mathcal{G}$)
⟨19⟩ $[\cdot, \Gamma_2, \cdot, \cdot]$:= TransStep($s_i, \mathcal{P}$, exit($\ell, \mathcal{P}$), 1)
⟨20⟩ **if** ($\Gamma_2 \not\equiv \emptyset$) $\mathcal{G}$:= Build($\Gamma_2, \mathcal{P}, \mathcal{G}$)
⟨21⟩ $s_i \xrightarrow{\Delta_1} s_i'$ /* Execute abstract transition Δ_1 */
⟨22⟩ $s_i := s_i'$
⟨23⟩ $[\cdot, \Gamma_1, \Delta_1, \cdot]$:= TransStep($s_i, \mathcal{P}$, entry($\ell, \mathcal{P}$), 1)
endwhile
⟨24⟩ $[\cdot, \Gamma_2, \cdot, \cdot]$:= TransStep($s_i, \mathcal{P}$, exit($\ell, \mathcal{P}$), 1)
⟨25⟩ $\mathcal{G}$:= Build($\Gamma_2, \mathcal{P}, \mathcal{G}$)
⟨26⟩ S := Summarize($s, \mathcal{G}, 2$) /* Phase 2 */
⟨27⟩ **else** S := TransStep($s, \mathcal{P}$, outgoing($\ell, \mathcal{P}$), phase)
⟨28⟩ **if** (phase $\equiv 2$) $\overline{\mathrm{S}}$:= Modification of S
taking into account the information computed in phase 1
⟨29⟩ **else** $\overline{\mathrm{S}}$:= S
⟨30⟩ memo and **return** $\overline{\mathrm{S}}$

Figure 4.11: Two-phase Symbolic Simulation Algorithm

then returned. We have demonstrated this process in Section 4.4.

We proceed to phase 2 with this new transition system, $\mathcal{G}$, as in line 4. The worst case bound is then achieved by looking for the maximum value in all returned solution paths Γ (line 5).

Our *key* function, Summarize, takes as inputs a symbolic state $s \equiv \langle \ell, [\![s]\!] \rangle$, a

transition system, and a flag indicating which phase it is in. It then performs the analysis using the context $[\![s]\!]$ and returns the summarization for the program point ℓ as in Def. 23 in Section 4.3.

Base Cases: Summarize handles 4 base cases. First, when the symbolic state s is infeasible (line 6), no execution needs to be considered. Note that here path-sensitivity plays a role since only provably executable paths will be considered. Second, s is a terminal state (line 7). Here Id refers to the identity function, which keep the values of variables unchanged. Ending point of a loop is treated similarly in the third base case (line 8). The last base case, lines 9- 10, is the case that a summarization can be reused. We have demonstrated this step, with examples, in Section 4.4.

Expanding to next Program Points: Line 27 depicts the case when transitions can be taken from current program point ℓ, and ℓ is not a loop starting point. Here we call TransStep to move recursively to next program points. TransStep implements the traversal of transition steps emanating from ℓ, denoted by $\mathsf{outgoing}(\ell, \mathcal{P})$, by calling Summarize recursively and then compounds the returned summarizations into a summarization of ℓ. The inputs of TransStep are symbolic state s, the transition system $\mathcal{P}$, a set of outgoing transitions *TransSet* to be explored, and the current phase the algorithm is in.

For each t in *TransSet*, TransStep extends the current state with the transition. Resulting child state is then given as an argument in a recursive call to Summarize (line 34). From each summarization of a child returned by the call to Summarize, the algorithm computes a component summarization, contributed by that particular child to the parent as in lines 35-39. All of such components will be compounded using the JoinHorizontal function (line 40).

Note that the interpolant for the child state is propagated back to its parent using the *precondition operation* pre, where $\mathsf{pre}(t, \overline{\Psi})$ denotes the precondition of the postcondition $\overline{\Psi}$ wrt. the transition t. In an ideal case, we would want this

```
        function TransStep(s, P, TransSet, phase)
               Let s be ⟨ℓ, [[s]]⟩
⟨31⟩           S̄ := [ℓ, ∅, false, true]
⟨32⟩           foreach (t ∈ TransSet ∧ t contains r := r+α) do
⟨33⟩               s --t--> s′
⟨34⟩               [ℓ′, Γ, Δ, Ψ̄] := Summarize(s′, P, phase)
⟨35⟩               Γ′ := ∅
⟨36⟩               foreach (⟨r1, δ1⟩ ∈ Γ) do
⟨37⟩                   one := {⟨r1 + α, combine(t, δ1)⟩}
⟨38⟩                   Γ′ := Merge_Paths(Γ′, one)
                   endfor
⟨39⟩               S := [ℓ, Γ′, combine(t, Δ), pre(t, Ψ̄)]
⟨40⟩               S̄ := JoinHorizontal(S̄, S)
               endfor
⟨41⟩           return S̄
```

operation to return the *weakest precondition*. But in general that could be too expensive. Discussions on possible implementations of this operator can be found at [Rybalchenko and Sofronie-Stokkermans, 2007; Chu and Jaffar, 2011]. In our implementation using CLP($\mathcal{R}$) [Jaffar *et al.*, 1992], the combine function simply conjoins the corresponding constraints and performs projections to reduce the size of the formula.

Loop Handling: Lines 13-26 handle the case when the current program point ℓ is the loop entry point. Let $\mathsf{entry}(\ell, \mathcal{P})$ denote the set of transitions going into the body of the loop, and $\mathsf{exit}(\ell, \mathcal{P})$ denote the set of transitions exiting the loop.

Upon encountering a loop, our algorithm attempts to unroll it once by calling the function TransStep to explore the entry transitions (line 16). When the returned set of representative paths is empty, it means that we cannot go into the loop body anymore, we thus proceed to the exit transitions (lines 24-25). Otherwise, if some feasible paths are found by going into the loop body, we first use the returned set of representative paths Γ_1 to add new transitions into our transition system $\mathcal{G}$ (line 18). Next we use the returned abstract transformer to produce a new continuation con-

text (lines 21-22), so that we can continue the analysis with the next iteration. Here we assume that this unrolling process will eventually terminate. However, since we are performing symbolic execution, it is possible that at some iterations both the entry transitions and exit transitions are feasible. Lines 19-20 accommodate this fact.

Phase 2: In phase 2, we now make use of assertions, to block paths. This is achieved at lines 11-12. Note that the negation of the assertion will be the interpolant for the current state and this interpolant will be propagated backward.

The summarization of a program point ℓ at phase 2 will be modified, as in line 28. The abstract transformer of this summarization is the one computed in phase 1. However, the interpolant is combined by conjoining the interpolant of that program point already computed in phase 1 to the current interpolant in phase 2. This is because an interpolant of a node comes from two sources. The first is due the *infeasible paths* detected in phase 1, while the second is due to the *blocked paths* detected in phase 2. We note here that, for simplicity, we purposely omit the details in phase 1 on how summarizations are vertically combined, so as to produce a serialization of summarization for the loop entry point. See Chapter 3 for details.

Merge_Paths and JoinHorizontal: Function Merge_Paths simply merges two sets of paths into one. As mentioned before, for each distinct way of changing the frequency variables which are relevant to some assertions used later, we only keep the dominating path and ignore all the dominated paths.

Given two subtrees T_1 and T_2 which are siblings and the inputs S_1 and S_2 summarize T_1 and T_2 respectively. JoinHorizontal is then used to produce the summarization of the compounded subtree T of both T_1 and T_2. Here the representative paths are merged (line 52). Preserving all infeasible/blocked paths in T requires preserving infeasible/blocked paths in both T_1 and T_2 (line 54). The input-output relationship of T is safely abstracted as the disjunction of the input-output relationships of T_1 and T_2 respectively (line 53). In our implementation, this corresponds to the convex

```
function Merge_Paths(Γ1, Γ2)
⟨42⟩  Γ := Γ1
⟨43⟩  foreach (γ2 :=⟨r2, δ2(c̃, c̃′)⟩ ∈ Γ2) do
⟨44⟩     status := true
⟨45⟩     foreach ( γ1 :=⟨r1, δ1(c̃, c̃′)⟩ ∈ Γ) do
⟨46⟩        if ( δ1(c̃, c̃′) ≡^A δ2(c̃, c̃′))
⟨47⟩           status := false
⟨48⟩           if (r2 > r1) replace γ1 in Γ by γ2
⟨49⟩           break                          /* Out of the inner loop */
         endfor
⟨50⟩     if (status) add γ2 into Γ
      endfor
⟨51⟩  return Γ
function JoinHorizontal(S1, S2)
      Let S1 be [ℓ, Γ1, Δ1, Ψ̄1]
      Let S2 be [ℓ, Γ2, Δ2, Ψ̄2]
⟨52⟩  Γ := Merge_Paths(Γ1, Γ2)
⟨53⟩  Δ := Δ1 ∨ Δ2                /* Merge two abstract transformers */
⟨54⟩  Ψ̄ := Ψ̄1 ∧ Ψ̄2                       /* Conjoin two interpolants */
⟨55⟩  return [ℓ, Γ, Δ, Ψ̄]
```

hull operator of the polyhedral domain.

4.5.1 Extension to Non-Cumulative Resource Analysis

As our framework is path-based, to extend it to work with *non-cumulative* resource is relatively easy. For non-cumulative resource, we need to capture not only the *net usage* but also the *high watermark* usage of that resource. For each class which modifies the frequency variables in an distinct way, we now consider dominating value not only in term of the net usage, but also in term of *high watermark* usage.

Consequently, for each of such class, a representative path is chosen ($\gamma \in \Gamma$), and is now of the form $\langle \mathsf{r}_n, \mathsf{r}_h, \delta(\tilde{c}, \tilde{c}')\rangle$, where $\delta(\tilde{c}, \tilde{c}')$, as before, captures how that class modifies the set of (relevant) frequency variables, r_n is the highest net amount of resource consumed in that class, r_h is the highest spike in term of high watermark observed by that class.

EXAMPLE 4.7 : Let us have a look at the following example, where m captures the amount of consumed memory. This code fragment contains no frequency variables.

$\langle 1 \rangle$ `if (*) { m += 100; m -= 90; }`
`else { m += 40; m -=10; }` $\langle 2 \rangle$

Now the first path (**then** branch) leads to an increase of 10 for the *net* memory usage while the second path (**else** branch) leads to an increase of 30. On the other hand, the first path leads to a spike of 100 in term of the *high watermark* usage while the second path gives rise to a spike of 40 only. As such, in summarizing this block, Γ will be $\{\langle 30, 100, \mathsf{Id}(\tilde{c})\rangle\}$. $\mathsf{Id}(\tilde{c})$ captures the fact that this class (of paths) does not change any frequency variable. Note that the two worst-case values came from different concrete paths. The compact representation for this code fragment would be $\langle\langle 1\rangle, \mathsf{spike}(100), \mathsf{m} := \mathsf{m}{+}30, \langle 2\rangle\rangle$.

Here we extend the notation for a transition to include also predicate $\mathsf{spike}(100)$, which indicates the fact that from program point $\langle 1\rangle$ there is a spike of 100 in term of the memory high watermark. If in some context, we come to the same program point $\langle 1\rangle$ with a net memory usage of 20, this will lead an observation of 120 in the memory high watermark. Note also that the value for a spike is always non-negative, a `free` statement would give a spike of 0. This is observed right before executing the `free` statement. However, that `free` statement would give a negative amount in term of change in memory net usage.

The modification of our algorithm is quite intuitive, therefore, we will be brief and only show the *key* changes in the TransStep function in order to accommodate the newly introduced component r_h of each representative path. See Fig. 4.12. As we now work with non-cumulative resource, the amount of change in net memory usage can be negative, i.e., $\alpha < 0$. Also we assume all original transitions (coming directly from the program input) contain the predicate $\mathsf{spike}(0)$.

We now briefly illustrate on the memory high watermark example (Ex. 4.5) presented earlier in Fig. 4.8.

```
        function TransStep(s, P, TransSet, phase)
                Let s be ⟨ℓ, [[s]]⟩
⟨56⟩    S̄ := [ℓ, ∅, false, true]
⟨57⟩    foreach (t ∈ TransSet ∧ t contains r := r+α and spike(β)) do
⟨58⟩        s --t--> s′
⟨59⟩        [ℓ′, Γ, Δ, Ψ̄] := Summarize(s′, P, phase)
⟨60⟩        Γ′ := ∅
⟨61⟩        foreach (⟨r_n, r_h, δ⟩ ∈ Γ) do
⟨62⟩            one := {⟨r_n + α, max(spike(β), r_h + α), combine(t, δ)⟩}
⟨63⟩            Γ′ := Merge_Paths(Γ′, one)
            endfor
⟨64⟩        S := [ℓ, Γ′, combine(t, Δ), pre(t, Ψ̄)]
⟨65⟩        S̄ := JoinHorizontal(S̄, S)
        endfor
⟨66⟩    return S̄
```

Figure 4.12: TransStep for Non-Cumulative Resource

EXAMPLE 4.8 : Recall that as we cannot automatically reason about the external function `parity`, in all the loop iterations, both available paths are deemed as feasible. However, they modify the frequency variables c_1 and c_2 differently. Specifically, the first path increments c_1 and does not change c_2; while the second path does not change c_1 and increments c_2. Also note that both c_1 and c_2 are used in the provided assertion.

Consequently, in the first phase, for each iteration, both paths are kept in building the new transition system. Similar to Ex. 4.6, at the end of the first phase, we build a new transition system as below. Note that this loop is executed for 100 iterations, and the last transition corresponds to the exit of the loop.

$\langle\langle\langle 4\rangle\text{-}0\rangle, \mathsf{spike}(10), \mathsf{c}_1 := \mathsf{c}_1{+}1 \wedge \mathsf{m} := \mathsf{m}{+}10, \langle\langle 4\rangle\text{-}1\rangle\rangle$

$\langle\langle\langle 4\rangle\text{-}0\rangle, \mathsf{spike}(0), \mathsf{c}_2 := \mathsf{c}_2{+}1 \wedge \mathsf{m} := \mathsf{m}{-}10, \langle\langle 4\rangle\text{-}1\rangle\rangle$

...

$\langle\langle\langle 4\rangle\text{-}99\rangle, \mathsf{spike}(10), \mathsf{c}_1 := \mathsf{c}_1{+}1 \wedge \mathsf{m} := \mathsf{m}{+}10, \langle\langle 4\rangle\text{-}100\rangle\rangle$

$\langle\langle\langle 4\rangle\text{-}99\rangle, \mathsf{spike}(0), \mathsf{c}_2 := \mathsf{c}_2{+}1 \wedge \mathsf{m} := \mathsf{m}{-}10, \langle\langle 4\rangle\text{-}100\rangle\rangle$

$\langle\langle\langle 4\rangle\text{-}100\rangle, \mathsf{spike}(0), , \langle 12\rangle\rangle$

We start phase 2 on this new transition system, using the initial context ($m = 10, c_1 = 0, c_2 = 0$). Our provided assertion, `assert(|c1 - c2| <= 1)` will be checked at every program point $\langle\langle 4\rangle\text{-j}\rangle$, for $j = 1..100$. Making use of the assertion, at the end of phase 2, the representative path computed for program point $\langle\langle 4\rangle\text{-}0\rangle$ (using new TransStep function) is $\langle 0, 10, \mathsf{c}_1 := \mathsf{c}_1{+}50 \wedge \mathsf{c}_2 := \mathsf{c}_2{+}50\rangle$. It means that there is a maximum spike of 10 and the change in net memory usage of the whole loop is 0. As we start with ($m = 10, c_1 = 0, c_2 = 0$) at $\langle\langle 4\rangle\text{-}0\rangle$, 20 is the bound for the memory high watermark. Note that, however, all the non-blocked paths end up having the net memory usage of 10 at the end of the loop.

4.5.2 Correctness Statements

We conclude this Section with two statements. The first says that the analysis produced by the algorithm is correct over all execution traces that *satisfy* the assertions. The second says that given a program, global assertions (frequency variables not reset), and that we are pursuing a cumulative analysis, our algorithm is better than the classic IPET algorithm. Briefly, this is because the only component of our algorithm which is lossy is in phase 1, and here we suffer only from the possible inclusion of some infeasible path satisfying the assertions. Such a path cannot be excluded by the IPET method.

Theorem 2. *The algorithm is* sound *wrt. assertions.*

Theorem 3. *The algorithm is* uniformly better *than the IPET algorithm.*

4.6 Experimental Evaluation

We used a 2.93Gz Intel processor and 2GB RAM. Our prototype is built upon CLP($\mathcal{R}$)[Jaffar *et al.*, 1992] and its native constraint solver.

The programs used for evaluation are: (1) academic examples presented in this Chapter; (2) benchmarks from Mälardalen WCET group [Mälardalen, 2006], which

Benchmark	LOC	Path-Sensitive				Path-Insensitive (IPET)	
		w.o. Assertions		w. Assertions		w.o. As	w. As
		Bound	T(s)	Bound	T(s)		
sparse_array (Ex-4.3)	< 100	110404	1.50	10404	3.48	110404	10404
bubblesort100 (Ex-4.4)	< 100	515398	5.52	**49798**	11.45	1019902	1019902
watermark (Ex-4.5)	< 100	1010	1.74	**20**	5.45	*	*
conflict100 (Ex-4.6)	< 100	1504	3.47	**759**	9.22	1504	1129
insertsort100	< 100	515794	4.91	**30802**	7.78	1020804	1020804
crc	128	1404	7.73	1084	8.61	1404	1084
expint	157	15709	4.40	**859**	4.56	-	-
matmult100	163	3080505	4.55	131705	5.54	3080505	131705
fir	276	1129	2.35	**793**	2.39	-	-
fft64	219	7933	5.52	**1733**	6.04	-	-
tcas	400	159	3.84	**81**	3.9	172	94
statemate	1276	2103	9.65	**1103**	9.73	2271	1271
nsichneu_small	2334	483	9.43	**383**	9.51	2559	2459

Table 4.1: Experiments with and without Assertions

are often used for evaluations of WCET path analysis techniques; and (3) a real traffic collision avoidance system `tcas`. Their corresponding sizes (LOC) are given in the second column of Table 4.1. Note that only watermark (Ex-4.5) is about memory high-watermark analysis, the rest are about WCET analysis.

Table 4.1 shows the experimental results. We evaluate the effects of assertions in our path-sensitive framework as well as in IPET, a path-insensitive framework. We remark that IPET is the current state-of-the-art in WCET path analysis and is used in most available WCET tools (e.g., [aiT,]).

The last two columns are about IPET. Recall that IPET *always* requires assertions on loop bounds in order to produce an answer. For programs where such information can be easily extracted from the code, we provide IPET such loop bounds. Loop bounds which must be dynamically computed (e.g., from unrolling) are not provided to IPET. As a result, IPET cannot produce bounds for some programs, indicated as '-'. For programs where IPET can successfully bound, IPET running time is always less than 1 second, so we do not tabulate those timings individually. IPET cannot

handle memory high-watermark analysis, therefore it cannot handle `Ex-4.5`. This is indicated as '*'.

The third and fourth columns show the bounds and running times using our unrolling framework without employing assertions. These results correspond to the results produced by the algorithm in Chapter 3, which is representative for the state-of-the-arts in loop unrolling. On the other hand, the fifth and sixth columns report the performance of the algorithm proposed in this Chapter, possessing path-sensitivity as well as being compliant with assertions. As expected, our algorithm produces the best bounds for all instances. Importantly, for most programs, it achieves more precise bounds which neither path-sensitivity alone nor user assertions alone can achieve.

Our algorithm scales well, even for programs with (nested) loops and of practical sizes. This is due to the use of compounded summarizations. From a close investigation, we see that our algorithm preserves the *superlinear* behavior observed in Chapter 3. Also, the cost of complying with assertions, i.e., the cost of phase 2, depends mainly on the assertions and the maximum number of iterations for the loop where the assertions are used. Such cost does not depend on the size of the input program, nor the size of the overall symbolic execution tree.

We also note that the cost for phase 2 in the `watermark` program is relatively high due to the use of our powerful interpolation method (precondition propagation), but in this case, does not give more reuse/reduction than a naive one. Addressing this issue is about balancing between different interpolation methods, and is out of scope of this work.

In summary, we have repeated on earlier experiments in Chapter 3 to demonstrate the superiority of having path-sensitivity, not considering assertions. Then we considered assertions, and demonstrated two things. Foremost is that our two-phase algorithm, which is new, can scale to practical sized programs. We also demonstrated along the way that assertions can influence the resource analysis in a

significant way.

4.7 Summary

We considered the problem of symbolic simulation of programs, where loops are unrolled, in the pursuit of resource analysis. A main requirement is that assertions may be used to limit the possible execution traces. The first phase of the algorithm performed symbolic simulation without consideration of assertions. From this, a skeletal transition system — which now only concerns the assertions, and is much simpler than the system corresponding to the original program — is produced. The second phase now determines the worst-case path in this simplified transition system. While this problem is an instance of an NP-hard problem (RCSP), we argue that the instances arising from assertions in program analysis lend itself to efficient solution using a dynamic programming plus interpolation approach. Finally, benchmarks clearly show that the algorithm can scale.

Part II

Safety Verification of Concurrent Programs

Chapter 5

Combining State Interpolation and Partial Order Reduction

> All I'm going to tell you is investigations, whether it be this and others, where you have partial facts, analysts, agents are always trying to interpret what those facts mean, extrapolate from them what they mean.
>
> Robert Mueller

We consider the *state explosion problem* in safety verification of concurrent programs. This is caused by the interleavings of transitions from different processes. In explicit-state model checking, a general approach to counter this explosion is Partial Order Reduction (POR) (e.g., [Valmari, 1991; Godefroid, 1996]). This exploits the equivalence of interleavings of "independent" transitions: two transitions are independent if their consecutive occurrences in a trace can be swapped without changing the final state. In other words, POR-related methods prune away *redundant* process interleavings in a sense that, for each Mazurkiewicz [Mazurkiewicz,

1986] trace equivalence class of interleavings, if a representative has been checked, the remaining ones are regarded as redundant.

On the other hand, *symbolic execution* [King, 1976] is another method for program reasoning which recently has made increasing impact on software engineering research [Cadar *et al.*, 2011]. The main challenge for symbolic execution is the exponential number of symbolic paths. The works [Jaffar *et al.*, 2009; McMillan, 2010; Jaffar *et al.*, 2011] tackle successfully this fundamental problem by eliminating from the concrete model, on-the-fly, those facts which are *irrelevant* or *too-specific* for proving the unreachability of the error nodes. This learning phase consists of computing *state-based interpolants* in a similar spirit to that of conflict clause learning in SAT solvers.

Now symbolic execution with *state interpolation* (SI) has been shown to be effective for verifying sequential programs. In SI [Jaffar *et al.*, 2009; McMillan, 2010; Jaffar *et al.*, 2011], a node at program point ℓ in the reachability tree can be pruned, if its context is subsumed by the interpolant computed earlier for the same program point ℓ. Therefore, even in the best case scenario, the number of states explored by a SI method must still be at least the number of all *distinct* program points[1]. However, in the setting of concurrent programs, exploring each distinct global program point[2] once might already be considered prohibitive. In short, symbolic execution with SI *alone* is not efficient enough for the verification of concurrent programs.

Recent work [Yang *et al.*, 2008] has shown the usefulness of going *stateful* in implementing a POR method. It directly follows that SI can help to yield even better performance. In order to implement an efficient stateful algorithm, we are required to come up with an abstraction for each (concrete or symbolic) state. Unsurprisingly, SI often offers us good abstractions.

[1]Whereas POR-related methods do not suffer from this. Here we assume that the input concurrent program has already been preprocessed (e.g., by static slicing to remove irrelevant transitions, or by static block encodings) to reduce the size of the transition system for each process.

[2]The number of global points is the product of the numbers of local program points in all processes.

The above suggests that POR and SI can be very much *complementary* to each other. In this Chapter, we propose a general framework employing *symbolic execution* in the exploration of the state space, while both POR and SI are exploited for pruning. SI and POR are combined synergistically as the concept of interpolation. Interpolation is essentially a form of learning where the completed search of a *safe* subtree is then formulated as a recipe, ideally a *succinct* formula, for future pruning. The key distinction of our interpolation framework is that each recipe discovered by a node is *forced* to be conveyed back to its ancestors, which gives rise to pruning of larger subtrees.

In summary, we address the challenge: "combining classic POR methods with symbolic technique has proven to be difficult" [Kahlon *et al.*, 2009]. More specifically, we propose an algorithm schema to combine *synergistically* state interpolation with POR so that the sum is more than its parts. However, we first need to formalize POR wrt. a symbolic search framework with abstraction in such a way that: (1) POR can be *property driven* and (2) POR, or more precisely, the concept of persistent set, can be applicable for a set of states (rather than an individual state). While the main contribution is a theoretical framework, we also indicate a potential for the development of advanced implementations.

5.1 Related Work

Partial Order Reduction (POR) is a well-investigated technique in model checking of concurrent systems. Some notable early works are [Valmari, 1991; Godefroid, 1996]. Later refinements of POR, Dynamic [Flanagan and Godefroid, 2005] and Cartesian [Gueta *et al.*, 2007] POR (DPOR and CPOR respectively) improve traditional POR techniques by detecting collisions on-the-fly. These methods can, in general, achieve better reduction due to the more accurate detection of independent transitions.

One important weakness of traditional POR is that it is *insensitive* wrt. a target

safety property. In contrast, recent works have shown that property-aware reduction can be achieved by symbolic methods using a general-purpose SAT/SMT solver [Wang *et al.*, 2008b; Kahlon *et al.*, 2009; Wang *et al.*, 2009; Cordeiro and Fischer, 2011]. Verification is often encoded as a formula which is *satisfiable* `iff` there exists an interleaving execution of the programs that violates the property. Reductions happen inside the SAT solver through the addition of learned clauses derived by conflict analysis [Silva and Sakallah, 1996]. This type of reduction is similar to what we call *state interpolation* (SI).

The most relevant related work is [Kahlon *et al.*, 2009], which is the first to consider a combination of the POR and SMT. Subsequently, there was a follow-up work [Wang *et al.*, 2009].

In [Kahlon *et al.*, 2009], they began with an SMT encoding of the underlying transition system, and then they enhance this encoding with a concept of "monotonicity". The effect of this is that traces can be grouped into equivalence classes, and in each class, all traces which are *not monotonic* will be considered as *unsatisfiable* by the SMT solver. The idea of course is that such traces are in fact redundant. This work has demonstrated some promising results as most concurrency bugs in real applications have been found to be *shallow*.

However, there is a fundamental problem with scalability in [Kahlon *et al.*, 2009], as mentioned in the follow-up work [Wang *et al.*, 2009]: "It will not scale to the entire concurrent program" if we encode the whole search space as a single formula and submit it to an SMT solver.

Before describing [Wang *et al.*, 2009], we compare [Kahlon *et al.*, 2009] with our work. Essentially, the difference is twofold. First, in this Chapter, the theory for partial order reduction is *property driven*. In contrast, the monotonicity reduction of [Kahlon *et al.*, 2009] is not. We specifically exemplify the power of property driven POR in the later sections. Second, the encoding in [Kahlon *et al.*, 2009] is processed by a *black-box* SMT solver. Thus important algorithmic refinements are

not possible. Some examples:

- There are different options in implementing SI. Specifically in this work, we employ "precondition" computations. Using black-box solver, one has to rely on its fixed interpolation methods.

- Our approach is *lazy* in a sense that our solver is only required to consider *one* symbolic path at a time; in [Kahlon *et al.*, 2009] it is not the case. This matters most when the program is unsafe and finding counter-examples is relatively easy (there are many traces which violate the safety).

- In having a symbolic execution framework, one can direct the search process. This is useful since the order in which state interpolants are generated does give rise to different reductions. Of course, such manipulation of the search process is hard, if not impossible, when using a black-box solver.

In order to remedy the scalability issue of [Kahlon *et al.*, 2009], the work [Wang *et al.*, 2009] proposed a concurrent trace program (CTP) framework which employs both concrete execution and symbolic solving to strike balance between efficiency and scalability of SMT-based method. This approach is more appropriate for testing than for verification. The new direction of [Wang *et al.*, 2009], in avoiding the blow-up of the SMT solver, was in fact preceded by the work on under-approximation widening (UW) [Grumberg *et al.*, 2005]. As with CTP, UW models a subset, which will be incrementally enlarged, of all the possible interleavings as an SMT formula and submits it to an SMT solver. In UW the scheduling decisions are also encoded as constraints, so that the *unsatisfiable core* returned by the solver can then be used to further the search in probably a useful direction. This is the major contribution of UW. However, an important point is that this furthering of the search is a *repeated* call to the solver, this time with a weaker formula; which means that the problem at hand is now larger, having more traces to consider. On this repeated call, the work done for the original call is thus *duplicated*.

At first glance, it seems attractive and simple to encode the problem compactly as a set of constraints and delegate the search process to a general-purpose SMT solver. However, there are some fundamental disadvantages, and these arise mainly because it is thus hard to exploit the semantics of the program to direct the search inside the solver. This is fact evidenced in the related works mentioned above.

We believe, however, the foremost disadvantage of using a general-purpose solver lies in the encoding of process interleavings. For instance, even when a concurrent program has only *one* feasible execution trace, the encoding formula being fed to the solver is still of enormous size and can easily choke up the solver. More importantly, different from safety verification of sequential programs, the encoding of interleavings (e.g., [Kahlon *et al.*, 2009] uses the variable *sel* to model which process is selected for executing) often hampers the normal derivations of succinct conflict clauses by means of resolution. We empirically demonstrate the inefficiency of such approach in Sec. 5.7.

There have been other recent approaches addressing safety verification of concurrent programs. To name a few: [Gupta *et al.*, 2011] is based on CEGAR technology, [Albarghouthi *et al.*, 2010] is based on symbolic execution, while [Sinha and Wang, 2010; Sinha and Wang, 2011] are based on SMT. However, they digress from the POR family since they do not deal with the concept of swapping transitions.

5.2 Background and Discussions

We consider a concurrent system composed of a finite number of threads or processes performing atomic operations on shared variables. Let P_i $(1 \leq i \leq n)$ be a process with the set $trans_i$ of transitions. Assume that $trans_i$ contains no cycles. Even though loops are important in concurrent systems, for simplicity, we ignore them (because loop-free programs are sufficient to exhibit our main contributions and routine techniques, e.g., as from [Jaffar *et al.*, 2011], can always be employed to handle loops).

We also assume all processes have disjoint sets of transitions. Let $\mathcal{T} = \cup_{i=1}^{n} trans_i$ be the set of all transitions. Let V_i be the set of local variables of process P_i, and V_{shared} the set of shared variables of the given concurrent program. Let $pc_i \in V_i$ be a special variable representing the process program counter, and the tuple $\langle pc_1, pc_2 \cdots, pc_n \rangle$ represent the global program point. As before, let *SymStates* be the set of all global symbolic states of the given program where $s_0 \in SymStates$ is the initial state. A state $s \in SymStates$ comprises two parts: its *global program point* ℓ, also denoted by $\mathtt{pc}(s)$, which now is a tuple of local program counters, and its *symbolic constraint* $[\![s]\!]$ over the program variables. In other words, we denote a state s by $\langle \mathtt{pc}(s), [\![s]\!] \rangle$.

We consider the *transitions* of states induced by the program. Following [Godefroid, 1996], we only pay attention to *visible* transitions. A (visible) transition $t^{\{i\}}$ pertains to some process P_i. It transfers process P_i from control location ℓ_1 to ℓ_2. Recall that the application of $t^{\{i\}}$ will execute an operation op and we denote transition $t^{\{i\}}$ by $\ell_1 \xrightarrow{\mathsf{op}} \ell_2$. At some state $s \in SymStates$, when the ith component of $\mathtt{pc}(s)$, namely $\mathtt{pc}(s)[i]$, equals ℓ_1, we say that $t^{\{i\}}$ is *schedulable*[3] at s. And when s satisfies the guard $\mathtt{cond}$, denoted by $s \models \mathtt{cond}$, we say that $t^{\{i\}}$ is *enabled* at s. For each state s, let $Schedulable(s)$ and $Enabled(s)$ denote the set of transitions which respectively are schedulable at s and enabled at s. A state s, where $Schedulable(s) = \emptyset$, is called a *terminal state*.

Let $s \xrightarrow{t} s'$ denote transition step from s to s' via transition t. This step is possible only if t is *schedulable* at s. The effect of applying a transition t on a feasible state s to arrive at state s' is the same as in Chapter 2. However, for technical reasons needed below, we shall allow schedulable transitions emanating from an infeasible state; it follows that the destination state must also be *infeasible*.

EXAMPLE 5.1 : Consider two processes P_1, P_2: P_1 simply awaits for $x = 0$, while P_2 increments x. So each has one transition to transfer (locally) from control location

[3] This concept is not standard in traditional POR, we need it here since we are dealing with symbolic search.

$\langle 0 \rangle$ to $\langle 1 \rangle$. Assume that initially $x = 0$, i.e., the initial state s_0 is $\langle \langle 0,0 \rangle, x = 0 \rangle$. Running P_2 first we have the transition from state $\langle \langle 0,0 \rangle, x = 0 \rangle$ to state $\langle \langle 0,1 \rangle, x = 1 \rangle$. From here, we note that the transition from P_1 is now not enabled even though it is schedulable. If applied, it produces an infeasible (and terminal) state $\langle \langle 1,1 \rangle, x = 1 \wedge 1 = 0 \rangle$. Note that $(x = 1 \wedge 1 = 0) \equiv false$.

On the other hand, running P_1 first, we have the transition from $\langle \langle 0,0 \rangle, x = 0 \rangle$ to $\langle \langle 1,0 \rangle, x = 0 \rangle$. We may now have a subsequent transition step to $\langle \langle 1,1 \rangle, x = 1 \rangle$, which is a feasible terminal state.

For a sequence of transitions w (i.e., $w \in \mathcal{T}^*$), $Rng(w)$ denotes the set of transitions that appear in w. Also let $\mathcal{T}_\ell$ denote the set of all transitions which are schedulable somewhere after global program point ℓ. We note here that the schedulability of a transition at some state s only depends on the program point component of s, namely $\texttt{pc}(s)$. It does not depend on the constraint component of s, namely $[\![s]\!]$. Given $t_1, t_2 \in \mathcal{T}$ we say t_1 can *de-schedule* t_2 **iff** there exists a state s such that both t_1, t_2 are schedulable at s but t_2 is not schedulable after the execution of t_1 from s.

Following the above, $s_1 \stackrel{t_1 \cdots t_m}{\Longrightarrow} s_{m+1}$ denotes a sequence of state transitions, and as before, we say that s_{m+1} is reachable from s_1. We call $s_1 \stackrel{t_1}{\rightarrow} s_2 \stackrel{t_2}{\rightarrow} \cdots \stackrel{t_m}{\rightarrow} s_{m+1}$ a *feasible* derivation from state s_1, $\texttt{iff}$ $\forall\, 1 \leq i \leq m \bullet t_i$ is enabled at s_i. As mentioned earlier, an *infeasible* derivation results in an *infeasible state* (an infeasible state is still aware of its global program point). An infeasible state satisfies any safety property.

We define a *complete execution* trace, or simply trace, ρ as a sequence of transitions such that it is a derivation from s_0 and $s_0 \stackrel{\rho}{\Longrightarrow} s_f$ and s_f is a terminal state. A trace is infeasible if it is an infeasible derivation from s_0. If a trace is infeasible, then at some point, it takes a transition which is schedulable but is not enabled. From thereon, the subsequent states are infeasible states.

We follow the same definition for safety as in Chapter 2. In other words, the

given concurrent system is *safe* wrt. a safety property ψ if $\forall s \in \mathit{SymStates} \bullet$ if *s is reachable from the initial state s_0* then $s \models \psi$. A trace ρ is *safe* wrt. ψ, denoted as $\rho \models \psi$, if all its states satisfy ψ.

Partial Order Reduction

POR methods exploit the fact that paths often differ only in execution order of non-interacting or "independent" transitions. This notion of *independence* between transitions can be formalized by the following definitions [Godefroid, 1996]. Note here that traditional POR can only accommodate *concrete* states.

Definition 24 (Independence Relation)**.** *A symmetric relation $\mathcal{R} \subseteq \mathcal{T} \times \mathcal{T}$ is an* independence relation `iff` *for each $\langle t_1, t_2 \rangle \in \mathcal{R}$ the following properties hold for every state s:*

1. *if t_1 is enabled in s and $s \xrightarrow{t_1} s'$, then t_2 is enabled in s* `iff` *t_2 is enabled in s' ; and*

2. *if t_1 and t_2 are enabled in s, then there is a unique state s'' such that $s \stackrel{t_1 t_2}{\Longrightarrow} s''$ and $s \stackrel{t_2 t_1}{\Longrightarrow} s''$.* □

Definition 25 (Equivalence)**.** *Two traces are (Mazurkiewicz)* equivalent `iff` *one trace can be transformed into another by repeatedly swapping adjacent independent transitions.* □

Traditional algorithms employ the concept of trace equivalence for pruning. They operate as classic state space searches except that, at each encountered state s, they compute the subset T of the transitions enabled at s, and explore only the transitions in T. Intuitively, a subset T of the set of transitions enabled in a state s is called *persistent in s* if whatever one does from s, while remaining outside of T, does not interact with T. Formally, we have the following:

Definition 26 (Persistent)**.** *A set $T \subseteq \mathcal{T}$ of transitions enabled in a state s is* persistent in s `iff`*, for all feasible derivations $s \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_2 \ldots \xrightarrow{t_{m-1}} s_{m-1} \xrightarrow{t_m} s_m$*

including only transitions $t_i \in \mathcal{T}$ and $t_i \notin T$, $1 \leq i \leq m$, t_i is independent *with all the transitions in T.* □

Discussion

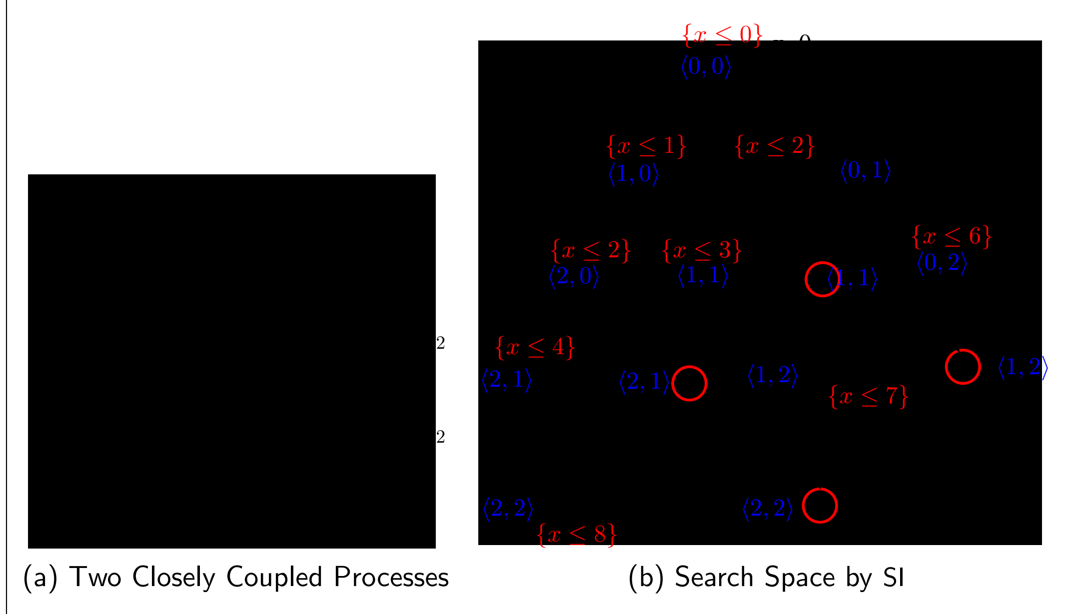
Now let us briefly illustrate the application of POR and SI to a simple example. We purposely make the example *concrete*, i.e., states are indeed concrete so that POR becomes applicable. This allows us to draw comparisons between POR and SI.

EXAMPLE 5.2 *(Closely coupled processes)*: See Fig. 5.1. Program points are shown in angle brackets. Fig. 5.1(a) shows the control flow graphs of two processes. Process 1 increments x twice whereas process 2 doubles x twice. The transitions associated with such actions and the safety property are depicted in the figure. POR requires a full search tree while Fig. 5.1(b) shows the search space explored by SI. Interpolants are in curly brackets. Bold circles denote pruned/subsumed states.

Let us first attempt this example using POR. It is clear that $t_1^{\{1\}}$ is *dependent* with both $t_1^{\{2\}}$ and $t_2^{\{2\}}$. Also $t_2^{\{1\}}$ is dependent with both $t_1^{\{2\}}$ and $t_2^{\{2\}}$. Indeed, each of all the 6 execution traces in the search tree ends at a different concrete state. As classic POR methods use the concept of *trace equivalence* for pruning, no interleaving is avoided: those methods will enumerate the full search tree of 19 states (for space reason, we omit it here).

Revisit the example using SI, where we use the *weakest preconditions* [Dijkstra, 1975] as the state interpolants: the interpolant for a state is computed as the weakest precondition to ensure that the state itself as well as all of its descendants are safe (see Fig. 5.1(b)). We in fact achieve the best case scenario with it: whenever we come to a program point which has been examined before, subsumption happens. The number of non-subsumed states is still of order $O(k^2)$ (where $k = 3$ in this particular example), assuming that we generalize the number of local program points for each process to $O(k)$. Fig. 5.1(b) shows 9 non-subsumed states and 4 subsumed states.

In summary, the above example shows that SI might outperform POR when



(a) Two Closely Coupled Processes (b) Search Space by SI

Figure 5.1: Application of SI on 2 Closely Coupled Processes

the component processes are closely coupled. However, one can easily devise an example where the component processes do not interfere with each other at all. Under such condition POR will require only one trace to prove safety, while SI is still (lower) bounded by the total number of global program points. In this Chapter, we contribute by proposing a framework to combine SI and POR *synergistically*.

5.3 State Interpolation Revisited

See Fig. 5.2. From s_0, by following one particular feasible sequence θ_1, we reach state s_i which is at global program point ℓ. Let $\mathcal{W}_\ell$ denote the set of all possible suffix traces starting from program point ℓ. For a sequence $w \in \mathcal{W}_\ell$, w may be a feasible or infeasible derivation from s_i.

We assume that the subtree A rooted at s_i has been explored and proved safe wrt. property ψ. The question now is: in subsequent (depth-first) traversal via θ_2 we reach state s_j which is also at the program point ℓ, can we avoid considering the subtree at s_j, namely subtree A'? If so, we consider this scenario as *state pruning*.

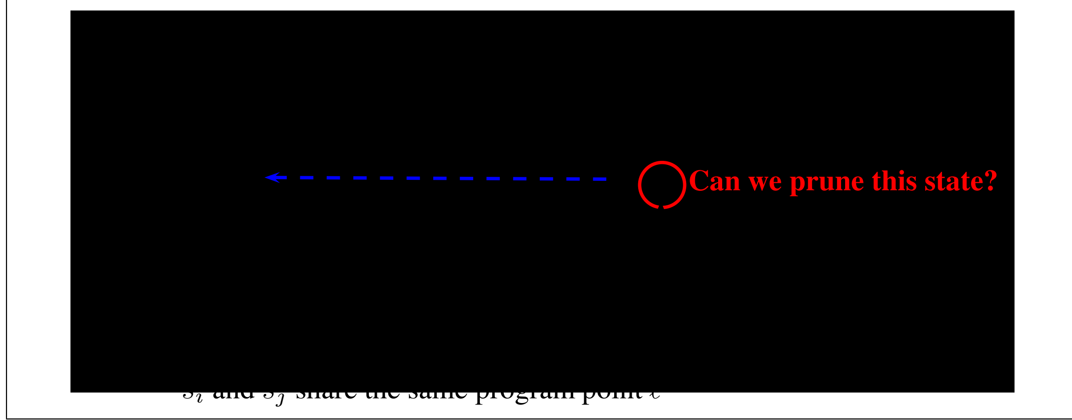


Figure 5.2: State Pruning

We now make the following definition which is crucial for the concept of pruning and will be used throughout this Chapter.

Definition 27 (Trace coverage). *Let ρ_1, ρ_2 be two traces of a concurrent system. We say ρ_1 covers ρ_2 wrt. a safety property ψ, denoted as $\rho_1 \sqsupseteq_\psi \rho_2$,* `iff` *$\rho_1 \models \psi \rightarrow \rho_2 \models \psi$.* □

One way to determine whether s_i covers s_j, i.e., state coverage, is to make use of the trace coverage concept. We can conclude that s_i covers s_j if $\forall w \in \mathcal{W}_\ell \bullet \theta_1 w \sqsupseteq_\psi \theta_2 w$. This seems elegant as it takes care of both feasible and infeasible traces. However, for some $w \in \mathcal{W}_\ell$, if $\theta_1 w$ is infeasible, i.e., at some point a symbolic *infeasible state* is reached, this trace will trivially satisfy ψ from that point on. It is then hard to ensure the safety of $\theta_2 w$ without exploring the subtree rooted at s_j. Instead, as in previous Chapters, during the exploration of subtree rooted at $s_i \equiv \langle \ell, [\![s_i]\!] \rangle$ we also compute a state-interpolant of s_i, denoted as $\mathsf{SI}(\ell, \psi)$, where $\mathsf{SI}(\ell, \psi)$ ensures that for all state $s_j \equiv \langle \ell, [\![s_j]\!] \rangle$ if $[\![s_j]\!] \models \mathsf{SI}(\ell, \psi)$ then $\forall t \in \mathit{Schedulable}(s_i)$[4] the two following conditions must be satisfied:

- if t was disabled at s_i, it also must be disabled at s_j

[4]Note that $\mathit{Schedulable}(s_i) = \mathit{Schedulable}(s_j)$.

- if t is a enabled at s_j (by the above condition, t must be enabled at s_i too) and $s_j \xrightarrow{t} s'_j$ and $s_i \xrightarrow{t} s'_i$, then s'_i must cover s'_j.

This observation enables us to compute the interpolants recursively. We do employ this idea for the state interpolation component of the synergy algorithm in Section 5.5.

5.4 Property Driven POR

"Combining classic POR methods with symbolic algorithms has been proven to be difficult" [Kahlon *et al.*, 2009]. The fundamental reason is that the concepts of (Mazurkiewicz) equivalence and transition independence, which drive all POR techniques, rely on the equivalence of two concrete states. However, in symbolic traversal, we rarely encounter two equivalent symbolic states.

It is even more difficult extending POR to be property driven. The difficulty arises from the fact that symbolic methods implicitly manipulate large *sets of states* as opposed to states individually. Capturing and exploiting transitions which are dynamically "independent" with respect to a set of states is thus much harder.

Let us next discuss about the *traditional* concept of transition independence. Here we ignore the matter of enablement and disablement of transitions. In order for two transitions t_1 and t_2 to be independent, it is required that for all state s in the state space, there is a unique state s' such that $s \overset{t_1 t_2}{\Longrightarrow} s'$ and $s \overset{t_2 t_1}{\Longrightarrow} s'$. Indeed, this requirement for the uniqueness of s', which in general is not satisfied in symbolic setting, hinders the extension of POR to symbolic techniques.

See Fig. 5.3. Assume that we have finished examining the subtree A, resulting from taking transition t_1 at state s_i. Now the question again is whether we can avoid those subtrees resulting from *not* taking t_1. Do we really need t_1 to be *independent* with those transitions appearing in the suffix subtree (in A)? In fact, the question becomes whether t_1 can *commute forward* over each of such transitions so that it

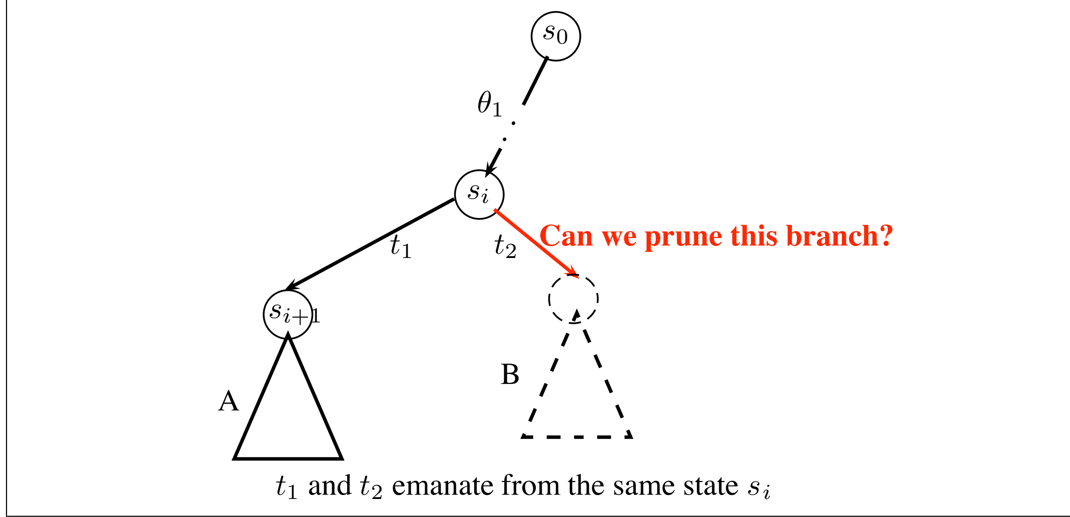


Figure 5.3: Branch Pruning

can blend in freely with those suffixes without affecting the safety. For example, we do not require t_2 to be able to commute forward over t_1 safely in order for the branch by taking transition t_2 to be pruned. Indeed, t_2, until executed, must still be schedulable in A (if not, we definitely cannot prune). In other words, in the traces ending at A, t_1 is always taken before t_2. Informally, the swapping is required to be one-way only.

Instead of using the concept of trace equivalence, from now on, we only make use of the concept of trace coverage. The concept of trace coverage is definitely weaker than the concept of Mazurkiewicz equivalence. In fact, if ρ_1 and ρ_2 are (Mazurkiewicz) equivalent then $\forall \psi \bullet \rho_1 \sqsupseteq_\psi \rho_2 \wedge \rho_2 \sqsupseteq_\psi \rho_1$. Now we will define a new and *weaker* concept which therefore generalizes the concept of transition independence.

Definition 28 (Semi-Commutative After A State)**.** *For a given concurrent program, a safety property ψ, and a derivation $s_0 \stackrel{\theta}{\Longrightarrow} s$, for all $t_1, t_2 \in \mathcal{T}$ which cannot de-schedule each other, we say t_1* semi-commutes *with t_2* after state *s wrt. $\sqsupseteq_\psi$, denoted by $\langle s, t_1 \uparrow t_2, \psi \rangle$,* `iff` *for all $w_1, w_2 \in \mathcal{T}^*$ such that $\theta w_1 t_1 t_2 w_2$ and $\theta w_1 t_2 t_1 w_2$ are execution traces of the program, then we have $\theta w_1 t_1 t_2 w_2 \sqsupseteq_\psi \theta w_1 t_2 t_1 w_2$.* □

From the definition, $Rng(\theta)$, $Rng(w_1)$, and $Rng(w_2)$ are pairwise disjoint. Importantly, if s is at program point ℓ, we have $Rng(w_1) \cup Rng(w_2) \subseteq \mathcal{T}_\ell \setminus \{t_1, t_2\}$. We observe that wrt. some ψ, if all important events, those have to do with the safety of the system, have already happened in the prefix θ, the "semi-commutative" relation is trivially satisfied. On the other hand, the remaining transitions might still interfere with each other (but not the safety), and do not satisfy the independent relation.

The concept of "semi-commutative" is obviously weaker than the concept of independence. If t_1 and t_2 are independent, it follows that $\forall \psi \; \forall s \bullet \langle s, t_1 \uparrow t_2, \psi \rangle \wedge \langle s, t_2 \uparrow t_1, \psi \rangle$. Also note that, in contrast to the relation of transition independence, but similar to the concept of *weak stubborn* [Valmari, 1991], the "semi-commutative" relation is *not symmetric*.

We now introduce a new definition for *persistent set*.

Definition 29 (Persistent Set Of A State). *A set $T \subseteq \mathcal{T}$ of transitions enabled in a state $s \in$ SymStates is* persistent in *s wrt. a property ψ* `iff`*, for all derivations $s \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_2 \ldots \xrightarrow{t_{m-1}} s_{m-1} \xrightarrow{t_m} s_m$ including only transitions $t_i \in \mathcal{T}$ and $t_i \notin T, 1 \leq i \leq m$, each transition in T* semi-commutes *with t_i after s wrt. $\sqsupseteq_\psi$.* $\square$

With the new definition of persistent set, we now can proceed with the normal *selective search* as described in classic POR techniques. In the algorithm presented in Fig. 5.4, we perform depth first search (DFS). For each state, computing a good persistent set from the "semi-commutative" relation is not a trivial task. Fortunately, the task is similar to computing the classical persistent set under the transition independence relation. The algorithms for this task can be found elsewhere (e.g., [Godefroid, 1996]).

Lemma 1. *The selective search algorithm in Fig. 5.4 is sound.*

Proof Outline. *Assume that there exist some traces which violate the property ψ and are not examined by our* selective search*. Let denote the set of such traces as*

```
Safety property ψ and initial state s0
⟨1⟩  Initially : Explore(s0)
function Explore(s)
⟨2⟩      if s ⊭ ψ
             Report Error and TERMINATE
⟨3⟩      T := Persistent_Set(s)
⟨4⟩      foreach t in T do
⟨5⟩          s --t--> s'          /* Execute t */
⟨6⟩          Explore(s')
⟨7⟩      endfor
end function
```

Figure 5.4: New Persistent-Set Selective Search (DFS)

$\mathcal{W}_{violated}$. *For each trace* $\rho = s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_2 \cdots \xrightarrow{t_m} s_m$, $\rho \in \mathcal{W}_{violated}$, *let* $first(\rho)$ *denote the smallest index* i *such that* t_i *is not in the persistent set of* s_{i-1}. *Without loss of generality, assume* $\rho_{max} = s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_2 \cdots \xrightarrow{t_m} s_m$ *having the maximum* $first(\rho)$. *Let* $i = first(\rho_{max}) < m$. *As the "commuter" and "commutee" cannot "de-schedule" each other, in the set* $\{t_{i+1} \cdots t_m\}$ *there must be a transition which belongs to the persistent set of* s_{i-1} *(otherwise, the must exist some transition that belongs to the persistent set of* s_{i-1} *which is schedulable at* s_m. *Therefore* s_m *is not a terminal state). Let* j *be the smallest index such that* t_j *belongs to the persistent set of* s_{i-1}. *By definition, wrt.* $\sqsupseteq_\psi$ *and after* s_{i-1}, t_j semi-commutes *with* $t_i, t_{i+1}, \cdots t_{j-1}$. *Also due to the definition of the "semi-commutative" relation we deduce that all the following traces (by making* t_j *repeatedly commute backward):*

$$\rho'_1 = t_1 t_2 \cdots t_{i-1} t_i t_{i+1} \cdots t_j t_{j-1} t_{j+1} \cdots t_m$$

$$\vdots$$

$$\rho'_{j-i-1} = t_1 t_2 \cdots t_{i-1} t_i t_j t_{i+1} \cdots t_{j-1} t_{j+1} \cdots t_m$$

$$\rho'_{j-i} = t_1 t_2 \cdots t_{i-1} t_j t_i t_{i+1} \cdots t_{j-1} t_{j+1} \cdots t_m$$

must violate the property ψ *too. However,* $first(\rho'_{j-i}) > first(\rho_{max})$. *This contradicts the definition of* ρ_{max}. □

In preparing for POR and SI to work together, we now further modify the concept

of persistent set so that it applies for a set of states sharing the same program point. The previous definitions apply for a specific state only.

Definition 30 (Semi-Commutative After A Program Point). *For a given concurrent program, a safety property ψ, and $t_1, t_2 \in \mathcal{T}$, we say t_1* semi-commutes *with t_2* after program point *ℓ wrt. $\sqsupseteq_\psi$ and ϕ, denoted as $\langle \ell, \phi, t_1 \uparrow t_2, \psi \rangle$,* `iff` *for all state $s \equiv \langle \ell, \cdot \rangle$ reachable from the initial state s_0, if $s \models \phi$ then t_1 semi-commutes with t_2 after state s wrt. $\sqsupseteq_\psi$.* □

Definition 31 (Persistent Set Of A Program Point). *A set $T \subseteq \mathcal{T}$ of transitions* schedulable *at program point ℓ is* persistent at *ℓ under the interpolant $\overline{\Psi}$ wrt. a property ψ* `iff`*, for all state $s \equiv \langle \ell, \cdot \rangle$ reachable from the initial state s_0, if $s \models \overline{\Psi}$ then for all derivations $s \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_2 \ldots \xrightarrow{t_{m-1}} s_{m-1} \xrightarrow{t_m} s_m$ including only transitions $t_i \in \mathcal{T}$ and $t_i \notin T, 1 \leq i \leq m$, each transition in T semi-commutes with t_i after state s wrt. $\sqsupseteq_\psi$.* □

Assume that $T = \{tp_1, tp_2, \cdots tp_k\}$. The interpolant $\overline{\Psi}$ can now be computed as $\overline{\Psi} = \bigwedge \phi_{ji}$ for $1 \leq j \leq k, 1 \leq i \leq m$ such that $\langle \ell, \phi_{ji}, tp_j \uparrow t_i, \psi \rangle$.

For each program point, it is possible to have different persistent sets associated with different interpolants accordingly. In general, a state which satisfies a stronger interpolant will have a smaller persistent set, and therefore, it enjoys more pruning.

5.4.1 Approximating the "semi-commutative" relation

In previous parts, we have defined the concepts of *trace coverage* and *"semi-commutative"*, which give rise to traditional selective search, thus exploit property driven pruning. To further relate these concepts to some practical implementation, we will hint out *one* possible way to approximate the "semi-commutative" relation, using state interpolation and precondition propagation.

Assume that we have at hand property ψ and two transitions $t_1, t_2 \in \mathcal{T}_\ell$. We now want to decide whether t_1 can semi-commute with t_2 after program point ℓ and wrt. $\sqsupseteq_\psi$. In fact, we pose the question as "with which ϕ we can have $\langle \ell, \phi, t_1 \uparrow t_2, \psi \rangle$".

Abusing notation, for each program point ℓ and a set of suffix traces $\mathcal{W}$ starting from ℓ, i.e., $\mathcal{W} \subseteq \mathcal{W}_\ell$, let $\texttt{SI}(\mathcal{W}, \psi)$[5] denote the state interpolant such that for all state $s \in \mathit{SymStates}$, s is at program point ℓ, if $s \models \texttt{SI}(\mathcal{W}, \psi)$ then all the derivations from s using the suffices in $\mathcal{W}$ are safe wrt. ψ. We also denote $\overline{\texttt{SI}}(\mathcal{W}, \psi)$ as the interpolant such that if $s \models \overline{\texttt{SI}}(\mathcal{W}, \psi)$, all the corresponding derivations are *unsafe*. Note that $\overline{\texttt{SI}}(\mathcal{W}, \psi) \models \texttt{SI}(\mathcal{W}, \neg\psi)$.

If $\mathcal{W}$ contains suffix traces starting from different program point, we define $\texttt{SI}(\mathcal{W}, \psi)$ as $\bigwedge_{i=1..m} \texttt{SI}(\mathcal{W}_i, \psi)$ where $\mathcal{W}_1 \cdots \mathcal{W}_m$ are partitions of $\mathcal{W}$ such that each set contains only suffices starting from the same program point. $\overline{\texttt{SI}}(\mathcal{W}, \psi)$ is defined similarly.

Now we can proceed by finding ϕ_1 and ϕ_2 such that:

$$\phi_1 = \overline{\texttt{SI}}(\{t_1 t_2 w_2 | \mathit{Rng}(w_2) \subseteq \mathcal{T}_\ell \setminus \{t_1, t_2\}\}, \psi)$$
$$\phi_2 = \texttt{SI}(\{t_2 t_1 w_2 | \mathit{Rng}(w_2) \subseteq \mathcal{T}_\ell \setminus \{t_1, t_2\}\}, \psi)$$

We adopt a "Hoare triple" [Hoare, 1969] notation here. We need to find ϕ such that for all possible sequences w_1, $\mathit{Rng}(w_1) \subseteq \mathcal{T}_\ell \setminus \{t_1, t_2\}$, we have $\{\phi\} w_1 \{\phi_1 \vee \phi_2\}$.

Assume we reach state s at program point ℓ through a prefix θ and $s \models \phi$. We then have $\forall w_1 \bullet s \stackrel{w_1}{\Longrightarrow} s_1$ *implies* $s_1 \models \phi_1 \vee \phi_2$. If it is unsafe before reaching s_1, trace coverage is trivially satisfied. Otherwise, $s_1 \models \phi_1 \vee \phi_2$ implies that $s_1 \models \phi_1 \vee s_1 \models \phi_2$. If $s_1 \models \phi_1$ then $\forall w_2 : \theta w_1 t_1 t_2 w_2$ must be unsafe, and therefore $\forall w_2 \bullet \theta w_1 t_1 t_2 w_2 \sqsupseteq_\psi \theta w_1 t_2 t_1 w_2$. Similarly, if $s_1 \models \phi_2$ then $\forall w_2 \bullet \theta w_1 t_2 t_1 w_2$ must be safe. Again we derive that $\forall w_2 \bullet \theta w_1 t_1 t_2 w_2 \sqsupseteq_\psi \theta w_1 t_2 t_1 w_2$.

5.5 Synergy of SI and PDPOR

We now show our combined framework. We assume for each program point, a persistent set and its associated interpolant are to be computed statically, i.e., by

[5] Note that with a proper $\mathcal{W}$, information about ℓ can always be inferred.

separate analyses. In other words, when we are at a program point, we can right away make use of the information about its persistent set.

The algorithm is in Fig. 5.5. The function Explore has input s_0 and assumes the safety property at hand is ψ. It naturally performs a depth first search of the state space.

```
Assume safety property ψ and initial state s0
⟨1⟩  Initially : Explore(s0)
function Explore(s)
       Let s be ⟨ℓ, ·⟩
⟨2⟩   if (memoed(s, Ψ̄)) return Ψ̄
⟨3⟩   if (s ⊭ ψ) Report Error and TERMINATE
⟨4⟩   Ψ̄ := ψ
⟨5⟩   ⟨T, Ψ̄_trace⟩ := Persistent_Set(ℓ)
⟨6⟩   if (s ⊨ Ψ̄_trace)
⟨7⟩      Ts := T
⟨8⟩      Ψ̄ := Ψ̄ ∧ Ψ̄_trace
⟨9⟩   else Ts := Schedulable(s)
⟨10⟩  foreach t in (Ts \ Enabled(s)) do
⟨11⟩          Ψ̄ := Ψ̄ ∧ pre(t, false)
⟨12⟩  endfor
⟨13⟩  foreach t in (Ts ∩ Enabled(s)) do
⟨14⟩     s --t--> s'                     /* Execute t */
⟨15⟩     Ψ̄' := Explore(s')
⟨16⟩     Ψ̄ := Ψ̄ ∧ pre(t, Ψ̄')
⟨17⟩  endfor
⟨18⟩  memo and return (Ψ̄)
end function
```

Figure 5.5: Algorithm Schema: A Framework for SI and POR (DFS)

Two Base Cases: The function Explore handles two base cases. One is when the current state is subsumed by some computed (and memoed) interpolant $\overline{\Psi}$. No further exploration is needed, and $\overline{\Psi}$ is returned as the interpolant (line 2). The second base case is when the current state is found to be *unsafe* (line 3).

Combining Interpolants: We make use of the (static) persistent set T computed for the current program point. We will further comment on this in the next section.

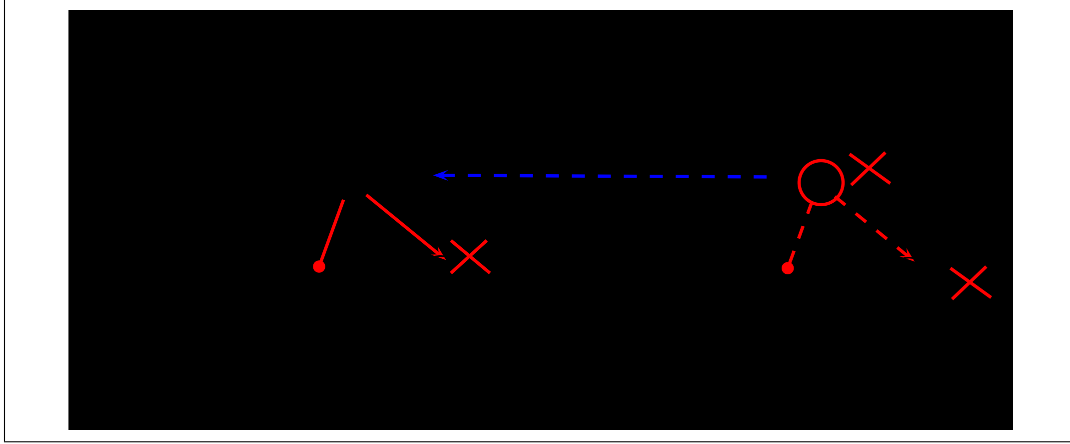


Figure 5.6: Inductive Correctness

The set of transitions to be considered is denoted by Ts. When the current state implies the interpolant $\overline{\Psi}_{trace}$ associated with T, we need to consider only those transitions in T. Otherwise, we need to consider all the schedulable transitions. Note that when the persistent set T is employed, the interpolant $\overline{\Psi}_{trace}$ must contribute to the combined interpolant of the current state (line 8). Disabled transitions (to be considered) at the current state will strengthen the interpolant as in line 11. Finally, we recursively (and selectively) follow those transitions which are enabled at the current state. The interpolant of each child state contributes to the interpolant of the current state as in line 16. In our framework, interpolants are propagated back using the precondition operation pre, where pre(t, ϕ) denotes a *safe approximation* of the weakest precondition wrt. the transition t and the postcondition ϕ [Dijkstra, 1975].

Theorem 4. *The synergy algorithm in Fig. 5.5 is sound.*

Proof Outline. *We use structural induction. Refer to Fig. 5.6. Assume that from* s_0 *we reach state* $s_i \equiv \langle \ell, \cdot \rangle$*. W.l.o.g., assume that at* s_i *there are three transitions which are schedulable, namely* t_1, t_2, t_3*, of which only* t_1 *and* t_2 *are enabled. Also assume that under the interpolant* $\overline{\Psi}_2$*, the persistent set of* ℓ*, and therefore of* s_i*, is just* $\{t_1\}$*. From the algorithm, we will extend* s_i *with* t_1 *(line 14) and attempt*

to verify the subtree A (line 15). Our induction hypothesis is that we have finished considering A, and indeed, it is safe under the interpolant $\overline{\Psi}_A$. That subtree will contribute $\overline{\Psi}_1 = \mathsf{pre}(t_1, \overline{\Psi}_A)$ (line 16) to the interpolant of s_i.

Using the interpolant $\overline{\Psi}_2$, the branch having transition t_2 followed by the subtree B is pruned and that is safe, due to Lemma 1. Also, the disabled transition t_3 contributes $\overline{\Psi}_3$ (line 11) to the interpolant of s_i. Now we need to prove that $\overline{\Psi} = \overline{\Psi}_1 \wedge \overline{\Psi}_2 \wedge \overline{\Psi}_3$ ***is indeed a sound interpolant for program point ℓ.***

Assume that subsequently in the search, we reach some state $s_j \equiv \langle \ell, \cdot \rangle$. We will prove that $\overline{\Psi}$ is a sound interpolant of ℓ by proving that if $s_j \models \overline{\Psi}$, then the pruning of s_j is safe.

First, $s_j \models \overline{\Psi}$ implies that $s_j \models \overline{\Psi}_3$. Therefore, at s_j, t_3 is also disabled. Second, assume that t_1 is enabled at s_j and $s_j \xrightarrow{t_1} s_{j+1}$ (if not the pruning of t_1 followed by A' is definitely safe). Similarly, $s_j \models \overline{\Psi}$ implies that $s_j \models \overline{\Psi}_1$. Consequently, $s_{j+1} \models \overline{\Psi}_A$ and therefore the subtree A' is safe too. Lastly, $s_j \models \overline{\Psi}$ implies that $s_j \models \overline{\Psi}_2$. Thus the reasons which ensure that the traces ending with subtree A cover the traces ending with subtree B also hold at s_j. That is, the traces ending with subtree A' also cover the traces ending with subtree B'. □

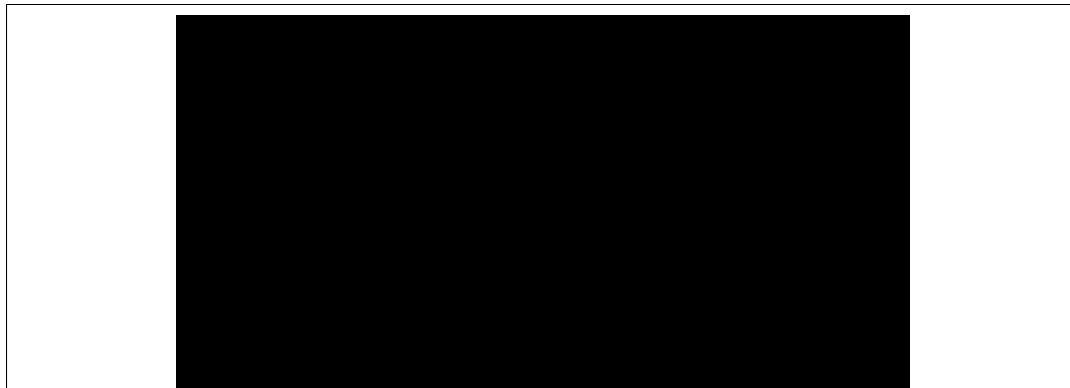


Figure 5.7: Two Producers and One Consumer

EXAMPLE 5.3 *(Two Producers and One Consumer)*: Fig. 5.7 shows the control flow graphs of three processes, which mimics the scenario of having two producers and one

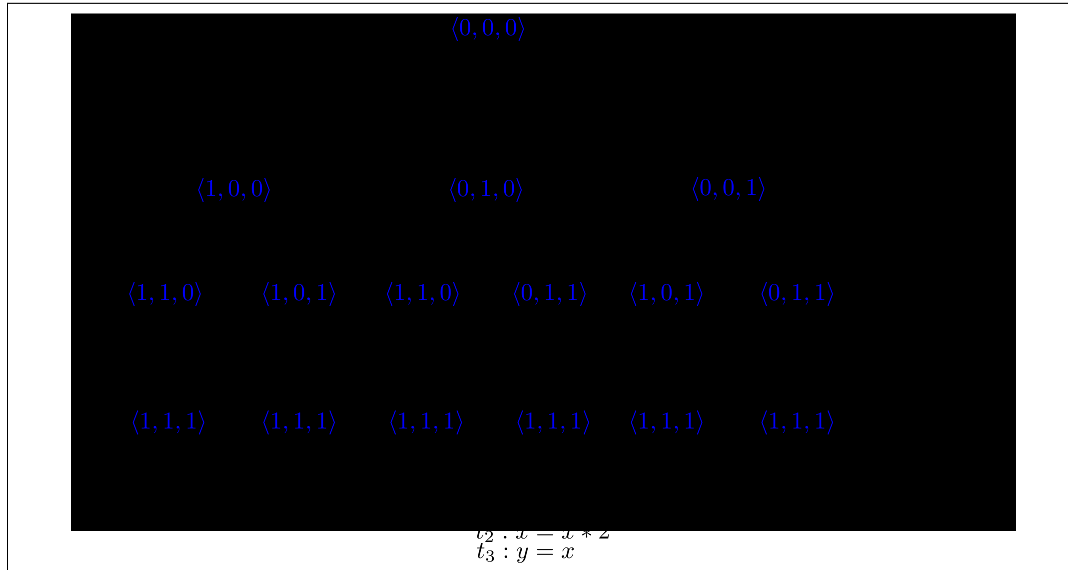


Figure 5.8: The Full Execution Tree

consumer in a concurrent system. Process 1 produces the value of shared variable x by incrementing it, while process 2 produces the value of x by doubling it. Process 3 consumes the value of x by assigning its value to y.

Pairwise, the processes interfere with each other. However, once the consumer has consumed the data, the producers no longer interfere with the correctness of the overall system. The safety property and relevant transitions are depicted in the figure. Although this example is very simple, it illustrates one of the most common (and general) type of data races in concurrent programs.

The full interleavings of the execution tree is shown in Fig. 5.8. The program is safe wrt. specified safety property ψ. POR (and DPOR)-only methods will enumerate the full execution tree which contains 16 states and 6 complete execution traces. Any technique which employs only the notion of Mazurkiewicz trace equivalence for pruning will have to consider at least 5 complete traces (due to 5 different terminal states). SI alone can reduce some states in this example; however, it will still explore at least 3 complete traces (and some partial traces of course). Symbolic methods as in [Grumberg *et al.*, 2005; Wang *et al.*, 2009] also will not perform well with this

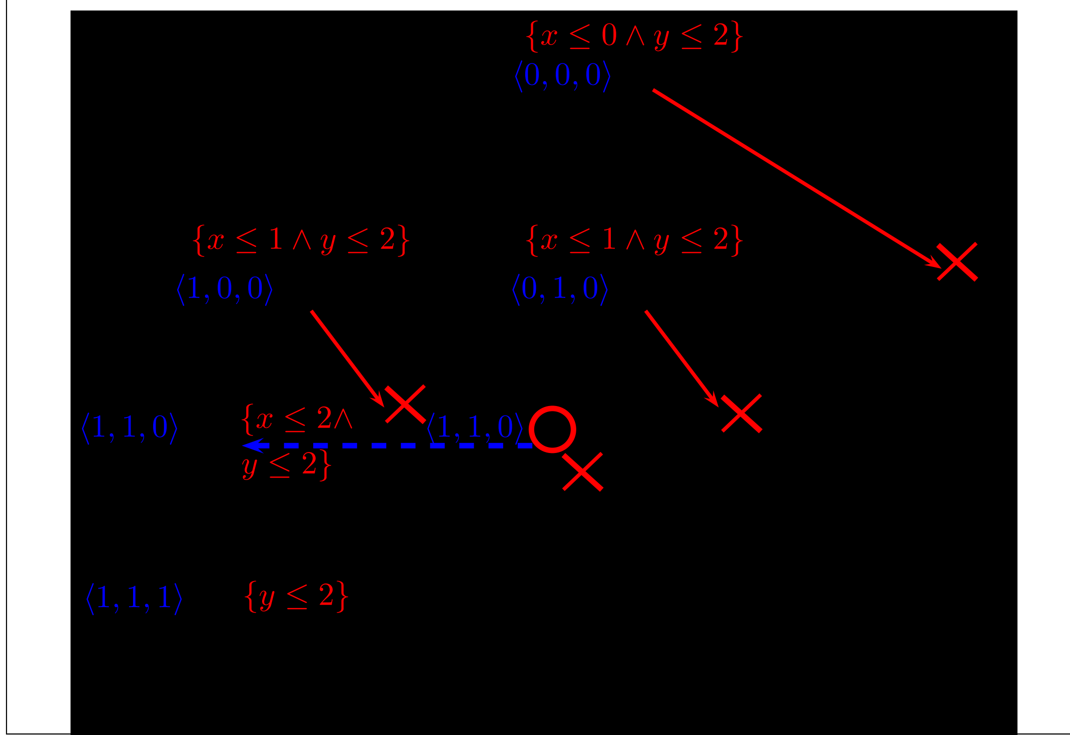


Figure 5.9: The Search Tree using Static Synergy Algorithm

example if we also consider the search space generated by solver.

Now we follow the algorithm described in Fig. 5.5. Fig. 5.9 illustrates the explored search space wrt. the following static knowledge (note that $\psi \equiv y \leq 2$):

$$\langle\langle 0{,}0{,}0\rangle, true, t_1 \ \uparrow \ t_3, \psi\rangle \tag{5.1}$$

$$\langle\langle 0{,}0{,}0\rangle, true, t_2 \ \uparrow \ t_3, \psi\rangle \tag{5.2}$$

$$\langle\langle 1{,}0{,}0\rangle, true, t_2 \ \uparrow \ t_3, \psi\rangle \tag{5.3}$$

$$\langle\langle 0{,}1{,}0\rangle, true, t_1 \ \uparrow \ t_3, \psi\rangle \tag{5.4}$$

Consider Fig. 5.9. At state s_0 of program point $\langle 0{,}0{,}0\rangle$ we have $\langle\langle 0{,}0{,}0\rangle, true, t_2 \ \uparrow t_3, \psi\rangle$ and $\langle\langle 0{,}0{,}0\rangle, true, t_1 \ \uparrow t_3, \psi\rangle$. Here we do not know whether t_1 can semi-commute with t_2 after program point $\langle 0{,}0{,}0\rangle$ under some condition. So we decide that the persistent set of $\langle 0{,}0{,}0\rangle$ is $\{t_1, t_2\}$ under the trace-interpolant $\overline{\Psi}_{\langle 0,0,0\rangle} \equiv true \ \wedge true \equiv$

true. Obviously $s_0 \models \overline{\Psi}_{\langle 0,0,0 \rangle}$, and as such the persistent set of s_0 is $\{t_1, t_2\}$.

Now assume that we follow t_1 reaching state s_1 at program point $\langle 1,0,0 \rangle$. Now in s_1 we have $x = 1, y = 0$. Now since $\langle \langle 1,0,0 \rangle, true, t_2 \uparrow t_3, \psi \rangle$, we decide that the persistent set of $\langle 1,0,0 \rangle$ is $\{t_2\}$ under the trace-interpolant $\overline{\Psi}_{\langle 1,0,0 \rangle} \equiv true$. Obviously $s_1 \models \overline{\Psi}_{\langle 1,0,0 \rangle}$, and as such the persistent set of s_1 is $\{t_2\}$.

Next we follow t_2 reaching program point $\langle 1,1,0 \rangle$ having $x = 2, y = 0$. Then we follow t_3 to complete a trace, reaching program point $\langle 1,1,1 \rangle$ with $x = 2, y = 2$. We backtrack and compute the interpolant for program point $\langle 1,1,0 \rangle$ which is $\langle \langle 1,1,0 \rangle, x \leq 2 \wedge y \leq 2 \rangle$. We continue to backtrack and compute the interpolant for program point $\langle 1,0,0 \rangle$ to be $x \leq 1 \wedge y \leq 2 \wedge true \equiv x \leq 1 \wedge y \leq 2$. Note that $x \leq 1$ is contributed by the propagation of the interpolant of $\langle 1,1,0 \rangle$, whereas *true* is contributed by the interpolant $\overline{\Psi}_{\langle 1,0,0 \rangle}$. Here the interleaving of t_3 from s_1 has been pruned due to the use of an interpolant.

Again, we backtrack to s_0 and continue the exploration with the transition t_2 (in the persistent set of s_0). Now we follow t_2 reaching state s_2 at program point $\langle 0,1,0 \rangle$, having $x = 0, y = 0$. At this program point we make use of $\langle \langle 0,1,0 \rangle, true, t_1 \uparrow t_3, \psi \rangle$. So we decide that the persistent set of $\langle 0,1,0 \rangle$ is $\{t_1\}$ under the trace-interpolant $\overline{\Psi}_{\langle 0,1,0 \rangle} \equiv true$. Obviously $s_2 \models \overline{\Psi}_{\langle 0,1,0 \rangle}$, as such the persistent set of s_2 is $\{t_1\}$.

Next from s_2 we follow the only transition in its persistent set, which is t_1, reaching the state s_3 at program point $\langle 1,1,0 \rangle$, having $x = 1, y = 0$. However, the program point $\langle 1,1,0 \rangle$ has been explored and proved safe with the interpolant $x \leq 2 \wedge y \leq 2$. As $s_3 \models x \leq 2 \wedge y \leq 2$, this state is then pruned.

In summary, at s_0, s_1, s_2, PDPOR helps to prune out transition t_3 whereas state interpolation helps us avoid the further exploration of s_3.

Let us try to go deeper in the process of discovering the "semi-commutative" relation. Using the law of algebra, that $\forall x \bullet x * 2 + 1 \leq (x + 1) * 2$, we can easily prove the facts mentioned above, namely (5.1), (5.2), (5.3), and (5.4).

Now let us explore the method discussed in Section 5.4.1. For instance, consider

(5.1) which is $\langle\langle 0{,}0{,}0\rangle, true, t_1 \uparrow t_3, \psi\rangle$. Let w_1, w_2 be sequences such that $w_1 t_1 t_3 w_2$ and $w_1 t_3 t_1 w_2$ are possible suffix traces from program point $\langle 0{,}0{,}0\rangle$. We know that $Rng(w_1)$ and $Rng(w_2)$ are disjoint and $Rng(w_1) \cup Rng(w_2) = \{t_2\}$. As t_2 does not modify the value of y, we can deduce that $Rng(w_2)$ contains no transition that can affect the safety property $\psi \equiv y \leq 2$ after t_3 had already executed. We have:

$\phi_1 = \overline{\mathtt{SI}}(\{x{+}{+}; y = x; w_2\}, \psi) \equiv x > 1 \vee y > 2$ and

$\phi_2 = \mathtt{SI}(\{y = x; x{+}{+}; w_2\}, \psi) \equiv x \leq 2 \wedge y \leq 2$.

Then $\phi_1 \vee \phi_2 \equiv true$. As $\forall \phi \bullet \{\phi\} w_1 \{true\}$, we can just use $\phi = true$ and achieve $\langle\langle 0{,}0{,}0\rangle, true, t_1 \uparrow t_3, \psi\rangle$.

We discover (5.2) in a similar manner. Furthermore, (5.3) (5.4) are easy to deduce as $\mathcal{T}_{\langle 0,1,0\rangle} \setminus \{t_1, t_3\} = \emptyset$ and $\mathcal{T}_{\langle 1,0,0\rangle} \setminus \{t_2, t_3\} = \emptyset$.

With the above example, we have demonstrated: (1) How the search space is explored by the synergy algorithm; and (2) How we can approximate the "semi-commutative" relation. One can also observe that our technique is incremental. Even when we cannot decide the "semi-commutative" relation for some transitions, the search space is still manageable. Moreover, though at the current state a transition cannot be pruned (by our PDPOR), there is still lots of potential for pruning at subsequent states and branches, by both SI and PDPOR.

5.6 Implementation of PDPOR

The challenge of implementing our proposed theory is essentially how to obtain a good estimate of the semi-commutative relation. Similarly to the traditional POR, the definitions are of paramount importance for the semantic use. In practice, however, one has to come up with sufficient conditions to efficiently implement the concepts. In our case, we estimate the semi-commutative relation in two steps:

1. We first employ *any* traditional POR method and first estimate the "semi-commutative" relation as the traditional independence relation (then the corresponding condition ϕ is just *true*). This is possible because the proposed

concepts are *strictly weaker* than the corresponding concepts used in traditional POR methods.

2. We then consider a number of scenarios in which we can prove that the semi-commutative relation holds. In fact, these scenarios suffice to deal with a number of important real-life applications.

Generally, we also want to detect the semi-commutative relation on the fly. This is sometimes possible if the state interpolants we computed are indeed the weakest preconditions. This step, however, is left as our future work.

To conclude this section, we demonstrate by presenting three common classes of problems, from which the semi-commutative relation, semantically, can be derived and proved.

Resource Usage of Concurrent Programs: Programs make use of limited resource (such as time, memory, bandwidth). Validation of resource usage in sequential setting is already a hard problem. It is even more challenging in the setting of concurrent programs, where the search space, due to interleavings, is astronomically huge.

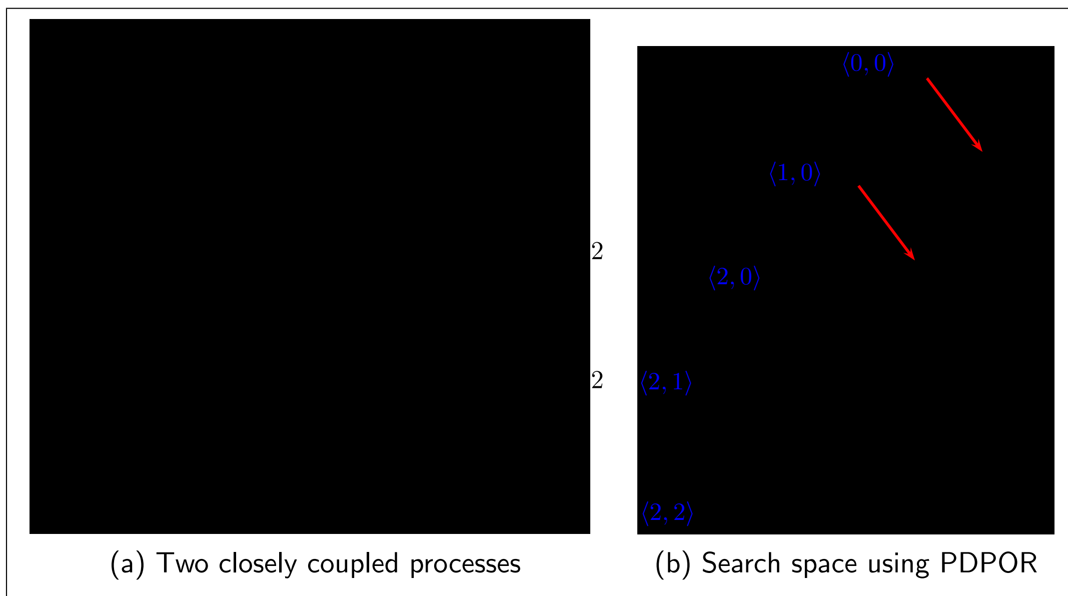
Here we model this class of problems by using a resource variable r. Initially, r is *zero*. Each process can increment or decrement variable r by some concrete value (e.g., memory allocation or deallocation respectively). A process can also double the value r (e.g., the whole memory is duplicated). However, the resource variable r cannot be used in the guard condition of any transition[6]. The property to be verified is that, "at all times, r is (upper-) bounded by some constant".

Proposition 1. *Let r be a resource variable of a concurrent program, and assume the safety property at hand is $\psi \equiv \mathsf{r} \leq C$, where C is a constant. For all transitions (assignment operations only) $t_1 : \mathsf{r} = \mathsf{r} + c_1$, $t_2 : \mathsf{r} = \mathsf{r} * 2$, $t_3 : \mathsf{r} = \mathsf{r} - c_2$ where $c_1, c_2 > 0$, we have for all program point ℓ:*

$$\langle \ell, true, t_1 \uparrow t_2, \psi \rangle \wedge \langle \ell, true, t_1 \uparrow t_3, \psi \rangle \wedge \langle \ell, true, t_2 \uparrow t_3, \psi \rangle \qquad \square$$

[6] As a result, we cannot model the behavior of a typical garbage collector.

Informally, other than common mathematical facts such as additions can commute and so do multiplications and subtractions, we also deduce that additions can semi-commute with both multiplications and subtractions while multiplications can semi-commute with subtractions. This Proposition can be proved by using basic laws of algebra.



(a) Two closely coupled processes (b) Search space using PDPOR

Figure 5.10: Example on performance of PDPOR

EXAMPLE 5.4 : Let us refer back to the example of two closely coupled processes introduced in Sec. 5.2. Fig. 5.10 shows again the control flow graphs of two processes, but now under the assumption that x is the resource variable of interest. We can clearly see that the program is safe wrt. safety property ψ. From Proposition 1, we need to explore only *one complete trace* to prove this safety.

In contrast, POR (and DPOR)-only methods will enumerate the full execution tree which contains 19 states and 6 complete execution traces. Any technique which employs only the notion of Mazurkiewicz trace equivalence for pruning will have to consider all 6 complete traces (due to 6 different terminal states). SI alone can reduce the search space in this example, and requires to explore only 9 states and

4 subsumed states (as in Sec. 5.2). Symbolic methods as in [Grumberg *et al.*, 2005; Wang *et al.*, 2009] also will not perform well with this example if we also consider the search space generated by solver. That is because a general solver has no mechanism to capture the fact that some transitions are relevant to the proof, though their orders are not.

Detection of Race Conditions: [Wang *et al.*, 2008a] proposed a property driven pruning algorithm to detect race conditions in multithreaded programs. This work has achieved more reduction in comparison with DPOR. The key observation is that, at a certain location (program point) ℓ, if their conservative "lockset analysis" shows that a search subspace is race-free, such search subspace can be pruned away. As we know, DPOR relies solely on the independence relation to prune redundant interleavings (if t_1, t_2 are independent, there is no need to flip their execution order). In [Wang *et al.*, 2008a], however, even when t_1, t_2 are dependent, we may skip the corresponding search space if flipping the order of t_1, t_2 does not affect the reachability of any race condition. In other words, [Wang *et al.*, 2008a] is indeed a (conservative) realization of our PDPOR, specifically targeted for detection of race conditions. Their mechanism to capture the such scenarios is by introducing a trace-based lockset analysis.

Ensuring Optimistic Concurrency: In the implementations of many concurrent protocols, *optimistic concurrency*, i.e., at least one process commits, is usually desirable. This can be modeled by introducing a flag variable which will be set when some process commits. The flag variable once set can not be unset. It is then easy to see that for all program point ℓ and transitions t_1, t_2, we have $\langle \ell, \mathsf{flag} = 1, t_1 \uparrow t_2, \psi \rangle$. Though simple, this observation will bring us more reduction compared to traditional POR methods.

5.7 Experiments

This section conveys two key messages. First, when state-based and trace-based methods are not effective individually, our combined framework still offers significant reduction. Second, property driven POR can be very effective, and applicable not only to academic programs, but also to programs used as benchmarks in the state-of-the-art.

We use a 3.2 GHz Intel processor and 2GB memory running Linux. Timeout is set at 10 minutes. In the tables, cells with '-' indicate timeout. We compare the performance of Partial Order Reduction alone (POR), State Interpolation alone (SI), the synergy of State Interpolation and Partial Order Reduction (SI+POR), i.e., the semi-commutative relation is estimated using only step one presented in Sec. 5.6, and when applicable, the synergy of State Interpolation and Property Driven Partial Order Reduction (SI+PDPOR), i.e., the semi-commutative relation is estimated using both steps presented in Sec. 5.6. For the POR component, we use the implementation from [Bokor *et al.*, 2011].

We start with parameterized versions of the *producer/consumer* example because its basic structure is extremely common. There are $2 * N$ producers and 1 consumer. Each producer will do its own non-interfered computation first, modeled by a transition which does not interfere with other processes. Then these producers will modify the shared variable x as follows: each of the first N producers increments x, while the other N producers double the value of x. On the other hand, the consumer consumes the value of x. The safety property is that the consumed value is no more than $N * 2^N$.

Table 5.1 is about this example. The performances presented in Table 5.1 clearly demonstrate the synergy benefits of POR and SI. SI+POR significantly outperforms both POR and SI. Note that this example can easily be translated to the resource usage problem, where our customized PDPOR requires only a *single* trace in order to prove safety.

	POR		SI		SI+POR		SI+PDPOR	
N	States	T(s)	States	T(s)	States	T(s)	States	T(s)
2	449	0.03	514	0.17	85	0.03	10	0.01
3	18745	2.73	6562	2.43	455	0.19	14	0.01
4	986418	586.00	76546	37.53	2313	1.07	18	0.01
5	–	–	–	–	11275	5.76	22	0.01
6	–	–	–	–	53261	34.50	26	0.01
7	–	–	–	–	245775	315.42	30	0.01

Table 5.1: Experiments on Producers/Consumer Example

To further demonstrate the power our synergy framework as well as the power of our property driven POR, we experiment next on the *Sum-of-ids* program. Here, each process (of N processes) has one unique *id* and will increment a shared variable *sum* by this *id*. We prove that in the end this variable will be incremented by the sum of all the ids.

Table 5.2 shows that POR does not result in any pruning for this benchmark. However, SI, and therefore SI+POR significantly prune the search space. Here we comment that the SI contribution is due to our specific interpolation algorithm; using a generic SMT solver would not achieve the same reduction. We confirm this by running the current version (4.1.2) of Z3 using the encodings presented in [Kahlon *et al.*, 2009]. The results do not scale. Finally, this example can also be translated to resource usage problem, our use of property-driven POR again requires *one* single trace to prove safety.

	POR = None		[Kahlon *et al.*, 2009] w. Z3			SI+POR = SI		SI+PDPOR	
N	States	T(s)	# Conflicts	# Decisions	T(s)	States	T(s)	States	T(s)
6	2676	0.44	1608	1795	0.08	193	0.05	7	0.01
8	149920	28.28	54512	59267	10.88	1025	0.27	9	0.01
10	–	–	–	–	–	5121	1.52	11	0.01
12	–	–	–	–	–	24577	8.80	13	0.01
14	–	–	–	–	–	114689	67.7	15	0.01

Table 5.2: Experiments on Sum-of-ids Example

We next use the parameterized version of the *dining philosophers*. We chose this

for two reasons. First, this is a classic example often used in concurrent algorithm design to illustrate synchronization issues and techniques for resolving them. Second, previous work [Kahlon *et al.*, 2009][7] has used this to demonstrate benefits from combining POR and SI.

The first safety property used in [Kahlon *et al.*, 2009], "it is not that all philosophers can eat simultaneously", is somewhat trivial. Therefore, here we verify a *tight* property, which is (a): "no more than *half* the philosophers can eat simultaneously". To demonstrate the power of symbolic execution, we verify this property *without* knowing the initial configurations of all the forks. Table 5.3 demonstrates the significant improvements of SI+POR over POR alone and SI alone. We note that the performance of our SI+POR algorithm is about 3 times faster than [Kahlon *et al.*, 2009].

	None		POR		SI		SI+POR	
Problem	States	T(s)	States	T(s)	States	T(s)	States	T(s)
`din-2(a)`	22	0.01	22	0.01	21	0.01	21	0.01
`din-3(a)`	1773	0.10	646	0.05	153	0.03	125	0.02
`din-4(a)`	–	–	155037	19.48	1001	0.17	647	0.09
`din-5(a)`	–	–	–	–	6113	1.01	4313	0.54
`din-6(a)`	–	–	–	–	35713	22.54	24201	4.16
`din-7(a)`	–	–	–	–	202369	215.63	133161	59.69
`bak-2`	86	0.05	48	0.03	38	0.03	31	0.02
`bak-3`	1755	3.13	1003	1.85	264	0.42	227	0.35
`bak-4`	47331	248.31	27582	145.78	1924	5.88	1678	4.95
`bak-5`	–	–	–	–	14235	73.69	12722	63.60

Table 5.3: Experiments on Dining Philosophers and Bakery Algorithm

We additionally considered a second safety property as in [Kahlon *et al.*, 2009], namely (b): "it is possible to reach a state in which all philosophers have eaten at least once". Our symbolic execution framework requires only a *single trace* (and less than 0.01 second) to prove this property in all instances, whereas [Kahlon *et al.*, 2009] requires even more time compared to proving property (a). This again

[7] [Kahlon *et al.*, 2009] is not publicly available. Therefore, it is not possible for us to make more comprehensive comparisons.

	ICSE11		SI		SI+PDPOR	
Problem	C	T(s)	States	T(s)	States	T(s)
micro_2	17	1095	20201	10.88	201	0.04
stack	12	225	529	0.26	529	0.26
circular_buffer	∞	477	29	0.03	29	0.03
stateful20	10	95	1681	1.13	41	0.01

Table 5.4: Experiments on Programs from ICSE11

illustrates the scalability issue of [Kahlon *et al.*, 2009], which is representative for other techniques employing general-purpose SMT solver for symbolic pruning.

To further highlight the power of symbolic execution and the synergy of POR and SI in our framework, we perform experiments on the "Bakery" algorithm. We note that, due to existence of infinite domain variables, model checking cannot handle Bakery algorithm. The results are also shown in Table 5.3.

Finally, we select four *safe* programs from [Cordeiro and Fischer, 2011] where the experimented methods did not perform well, namely micro_2, stack, circular_buffer, and stateful20. Table 5.4 shows the running time of SI alone and of the combined framework. For convenience, we also replicate the best running time reported in [Cordeiro and Fischer, 2011] and C is the context switch bound used. Because we assume no context switch bound, the corresponding value in our framework is ∞.

We can see that even our SI alone significantly outperforms the techniques in [Cordeiro and Fischer, 2011]. We believe it is due to the following reasons. First, our method is *lazy*, which means that only a path is considered at a time. [Cordeiro and Fischer, 2011] itself demonstrates the usefulness of this. Second, but importantly, we are *eager* in discovering infeasible paths. The program circular_buffer, which has only one feasible complete execution trace, therefore can be efficiently handled by our framework. This is one important advantage of symbolic execution, as discussed in [McMillan, 2010].

It is important to note that, PDPOR significantly improves the performance of SI wrt. programs micro_2 and stateful20. This further demonstrates the applicability

of our proposed framework.

5.8 Summary

We presented a framework for exploring and pruning the space of interleavings of a concurrent system, pursuant to a target property. In our framework, we first introduced a new notion of *property driven partial order reduction* in order to capture the reasoning capability of POR, but importantly, this time potentially in regard to the target property. We briefly showed how our theory on PDPOR can be customized for specific applications. But the main contribution is that the framework *combines* the use of trace-based reduction techniques with the more established notion of *state interpolant* used in general-purpose SMT solvers as well as CEGAR. This combination is *synergistic* in the sense that we obtain more pruning in the combination than if we had applied both state and PDPOR in separate phases.

Chapter 6

Complete Symmetry Reduction

> The most general law in nature is equity—the principle of balance and symmetry which guides the growth of forms along the lines of the greatest structural efficiency.
>
> Herbert Read

Symmetry reduction is a well-investigated technique to counter the state space explosion problem when dealing with concurrent systems whose processes are similar. In fact, traditional symmetry reduction techniques rely on an idealistic assumption that processes are *indistinguishable*. Because this assumption excludes many realistic systems, there is a recent trend [Emerson and Trefler, 1999; Emerson *et al.*, 2000; Sistla and Godefroid, 2004; Wahl, 2007; Wahl and D'Silva, 2010] to consider systems of non-identical processes, where the processes are *sufficiently similar* that the original gains of symmetry reduction can still be accomplished. However, this necessitates an intricate step of detecting symmetry in the state exploration.

We start by considering an intuitive notion of symmetry, which is based on a standard adaptation of the notion of bisimilarity. We say two states s_1 and s_2 are symmetric if there is a "permutation" π such that $s_2 = \pi(s_1)$, and if each successor

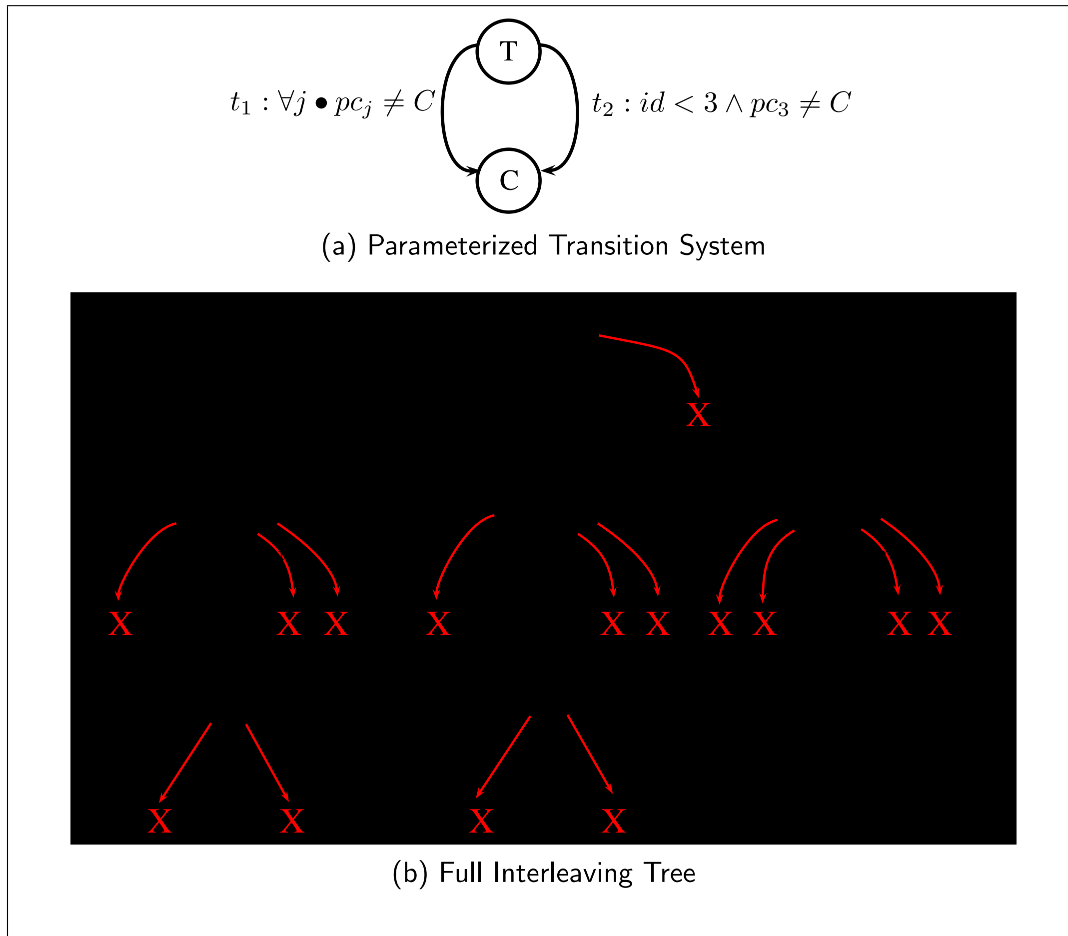
state of s_1 can be matched (via π) with a unique successor state of s_2 while at the same time each successor state of s_2 can be matched (via π^{-1}) with a unique successor state of s_1. In verification of a safety property ψ, we further require that ψ and $\pi(\psi)$ are equivalent.

We refer to this notion as *strong symmetry*. We mention that all recent works which deal with heterogeneous systems (processes are not necessarily identical) have the desire to capture this type of symmetry in the sense that they attempt, though not quite successfully, to consider only states which are *not* strongly symmetric to any already encountered state.

In this Chapter, we present a general approach to symmetry reduction for safety verification of a finite multi-process system, defined parametrically, without any prior knowledge about its global symmetry. In particular, we explicitly explore all possible interleavings of the concurrent transitions, while applying pruning on "symmetric" subtrees. We now introduce a new notion of symmetry: *weak symmetry*. Informally, this notion weakens the notion of permutation between states so that *the program counter* is the paramount factor in consideration of symmetry. In contrast, values of program variables are used in consideration of strong symmetry. The main result is that our approach is *complete* wrt. weak symmetry: it only considers states which are not weakly symmetric to an already encountered state.

In more details, we address the state explosion problem by employing *symbolic learning* on the search tree of all possible interleavings. Specifically, our work is based on the concept of interpolation. Here, interpolation is essentially a form of *backward learning* where a completed search of a *safe* subtree is then formulated as a recipe for pruning (every state/node is a root associated to some subtree). There are two key ideas regarding our learning technique: First, each learned recipe for a node not only can be used to prune other nodes having the same future (same program point), but also can be *transferred* to prune nodes that having *symmetric* futures (symmetric program points). Second, each recipe discovered by a node will

be conveyed back to its ancestors, which gives rise to pruning of *larger* subtree. Another important distinction is that our method learns *symbolically* with respect to the safety property and the interleavings. In Section 6.5, we will confirm the effectiveness of our method experimentally on some classic benchmarks.



(a) Parameterized Transition System

(b) Full Interleaving Tree

Figure 6.1: Modified 3-process Reader-Writer Protocol

Before proceeding, let us introduce two examples demonstrating the notions of strong and weak symmetry.

EXAMPLE 6.1 *(Modified Reader-Writer Protocol)*: We borrow this example, with modification, from [Wahl, 2007; Wahl and D'Silva, 2010] wherein are two "reader" processes (indices 1, 2) and one "writer" process (index 3). We denote by C and T the

local process states which indicate entering the critical section and in a "trying" state, respectively. See Fig. 6.1(a). Note that pc_j is the local control location of process j and for each process, id is its *process identifier*. These concepts will be defined more formally in Section 6.2.

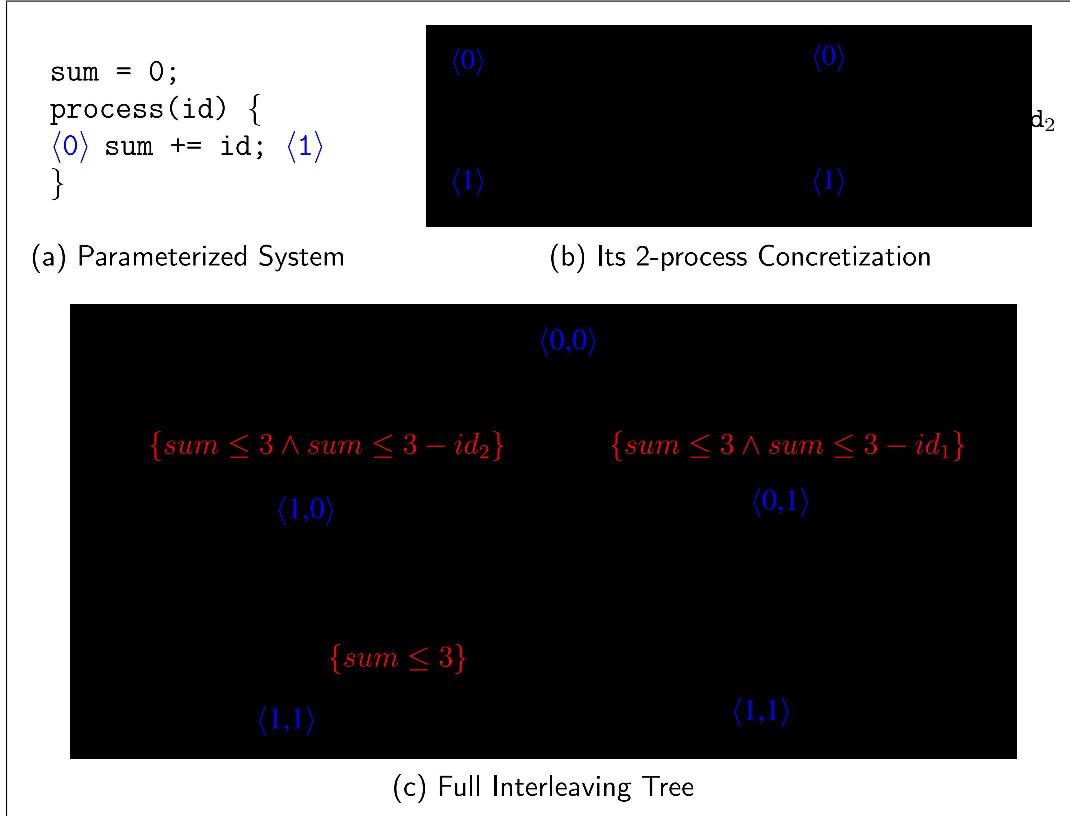
For each process, there are two transitions from T to C. The first, t_1, is executable by any process provided that no process is currently in its critical section ($\forall j \bullet pc_j \neq$ C). The second, t_2, is however available to only readers ($id < 3$), and the writer must be in a non-critical local state $pc_3 \neq$ C. This example shows symmetry between the reader processes, but because of their priority over the writer, we do not have "full" symmetry [Wahl, 2007].

Fig. 6.1(b) shows the full interleaving tree. Transitions are labelled with superscripts to indicate the process to which that transition is associated. *Infeasible* transitions are arrows ending with crosses. Note that nodes CTT and TCT are strongly symmetric, but neither is strongly symmetric with TTC.

EXAMPLE 6.2 *(Sum-of-ids)*: See Fig. 6.2(a) and Fig. 6.2(b). Initially, the shared variable `sum` is set to 0. There are two processes, each increments `sum` by the amount of its process identifier, namely `id`. The local transition systems for process 1 and process 2 are shown in Fig. 6.2(b). The full interleaving tree is shown in Fig. 6.2(c). Program points are in angle brackets. For clarity, we use $\langle 1,1 \rangle\#1$ and $\langle 1,1 \rangle\#2$ to denote the first and the second visit to the program point $\langle 1,1 \rangle$, respectively.

Let π be the function swapping the indices of the two processes. We can see that the subtrees rooted at states $\langle \langle 1,0 \rangle, sum = 1 \rangle$ and $\langle \langle 0,1 \rangle, sum = 2 \rangle$ share the same shape. However, due to the difference in the value of shared variable `sum`, strong symmetry does not apply (in fact, any top-down technique, such as [Wahl, 2007; Wahl and D'Silva, 2010; Sistla and Godefroid, 2004], cannot avoid exploring the subtree rooted at $\langle \langle 0,1 \rangle, sum = 2 \rangle$, even if the subtree rooted at $\langle \langle 1,0 \rangle, sum = 1 \rangle$ has been traversed and proved to be safe).

There is however a *weaker* notion of symmetry that does apply. We explain



(a) Parameterized System

(b) Its 2-process Concretization

(c) Full Interleaving Tree

Figure 6.2: Sum-of-ids Example

this by outlining our own approach, whose key feature is the computation of an *interpolant* [Jaffar *et al.*, 2009] for a node, by a process of backward learning. Informally, this interpolant represents a *generalization* of the values of the variables such that the traversed tree has a similar transition structure, and also remains safe. In the example, we require the safety property $\psi \equiv sum \leq 3$ at every state, and interpolants are shown as formulas inside curly brackets.

Using precondition propagation, the interpolant for state $\langle \langle 1{,}1 \rangle \#1, sum = 3 \rangle$ is computed as $sum \leq 3$, and the interpolant for state $\langle \langle 1{,}0 \rangle, sum = 1 \rangle$ is computed as $\overline{\Psi}_{\langle 1,0 \rangle} \equiv sum \leq 3 \wedge sum \leq 3 - id_2$. Using this, we can infer that $\overline{\Psi}_{\langle 0,1 \rangle} \equiv sum \leq 3 \wedge sum \leq 3 - id_1$ (obtained by applying π on $\overline{\Psi}_{\langle 1,0 \rangle}$) is a sound interpolant for program point $\langle 0{,}1 \rangle$. As $sum = 2 \models \overline{\Psi}_{\langle 0,1 \rangle}$, the subtree rooted at $\langle \langle 0{,}1 \rangle, sum = 2 \rangle$ can be pruned.

6.1 Related Work

Symmetry reduction has been extensively studied, e.g., [Emerson and Sistla, 1993; Clarke *et al.*, 1993; Ip and Dill, 1996; Emerson and Sistla, 1997]. Traditionally, symmetry is defined as a transition-preserving equivalence, where an automorphism π, other than being a bijection on the reachable states, also satisfies that (s, s') is a transition `iff` $(\pi(s), \pi(s'))$ is. There, this type of symmetry reduction is enforced by *unrealistic* assumptions about indistinguishable processes. As a result, it does not apply to many systems in practice.

One of the first to apply symmetry reduction strategies to "approximately symmetric" systems is [Emerson and Trefler, 1999], defining notions of *near* and *rough* symmetry. Near and rough symmetry is then generalized in [Emerson *et al.*, 2000] to *virtual symmetry*, which still makes use of the concept of bisimilarity for symmetry reduction. Though bisimilarity enables full μ-calculus model checking, the main limitation of these approaches is that they exclude many systems, where bisimilarity to the quotient is simply not attainable. Also, these approaches work only for the verification of *fully symmetric properties*. No implementation is provided.

The work [Sistla and Godefroid, 2004] allows arbitrary divergence from symmetry, and accounts for this divergence initially by conservative optimism, namely in the form of symmetric "super-structure". Specifically, transitions are added to the structure to achieve symmetry. A *guarded annotated quotient* (GAQ) is then obtained from the super-structure, where added transitions are marked. This approach works well for programs with syntacticly specified static transition priority. However, in general, the GAQ needs to be *unwound* frequently to compensate for the loss in precision (false positive due to added transitions). This might affect the running time significantly as this method might need to consider many combinations of transitions which do not belong to the original structure.

In comparison with our technique, this method has a clear advantage that it can handle arbitrary CTL* property. Nevertheless, our technique is more efficient

both in space and time. Our technique is required to store an interpolant for each non-subsumed state, whereas in [Sistla and Godefroid, 2004], a quotient edge might require multiple annotations. Furthermore, ours does not require a costly preprocessing of the program text to come up with a symmetric super-structure. Also, extending [Sistla and Godefroid, 2004] to symbolic model checking does not seem possible.

The most *recent* state-of-the-art regarding symmetry reduction, and also closest to our spirit, is the *lazy approach* proposed by [Wahl, 2007; Wahl and D'Silva, 2010]. Here only safety verification is considered. This approach does not assume any prior knowledge about (global) symmetry. Indeed, they initially and lazily ignore the potential lack of symmetry. During the exploration, each encountered state is annotated with information about how symmetry is violated along the path leading to it. The idea is that more similarity between component processes entails more compression is achieved.

In summary, the two main related works which are not restricted *a priori* on global symmetry are [Sistla and Godefroid, 2004] and [Wahl, 2007]. That is, these works allow the system to use process identifiers and therefore do not restrict the behaviors of individual processes. This is not the case with the previously mentioned works.

These works, [Sistla and Godefroid, 2004] and [Wahl, 2007], can be categorized as *top-down* techniques. Fundamentally, they look at the syntactic similarities between processes, and then come up with a reduced structure where symmetric states/nodes are merged into one abstract node. When model checking is performed, an abstract node might be concretized into a number of concrete nodes and each is checked one by one ([Sistla and Godefroid, 2004] handles that by unwinding). For them, two symmetric parental nodes are not guaranteed to have correspondingly symmetric children. For us, by backward learning, we *ensure* that is the case. Consequently, and most importantly, they do not exponentially improve the runtime, only compress

the state space.

Consider again Ex. 6.1 (Fig. 6.1). A top-down approach will consider TTC as a "potentially" symmetric state of CTT, and all three states CTT, TCT, and TTC are merged into one abstract state. While having compaction, it is not the case that the search space traversed is of this compact size. As a non-symmetric state (TTC) is merged with other mutually symmetric states (CTT and TCT), in generating the successor abstract state, the parent abstract state is required to be concretized and both transitions $t_2^{\{2\}}$ (emanating from CTT) and transition $t_2^{\{1\}}$ (emanating from TCT) are considered (in fact, infeasible transition $t_2^{\{3\}}$ is also considered). In summary, compaction may not lead to any reduction in the search space.

We finally mention that we consider only safety properties because we wish to employ abstraction in the search process. And it is precisely a judicious use of abstraction that enables us to obtain more pruning in comparison with prior techniques. We prove this in principle by showing that we are *complete* wrt. weak symmetry, and we demonstrate this experimentally on some classic benchmarks.

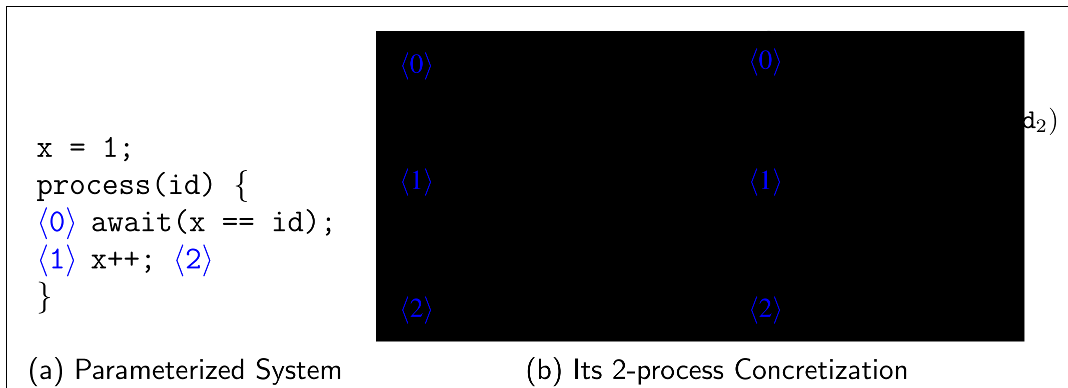
6.2 Preliminaries

We consider a parametrically defined n-process system, where n is fixed. In accordance with standard practice in works on symmetry, we assume that the domain of discourse of the program variables is *finite* so as to guarantee termination of the search process of the underlying transition system. Infinite domains may be accommodated by some use of abstraction, as we show in one benchmark example below.

We employ the usual syntax of a deterministic imperative language, and communication occurs via shared variables. Each process has a unique and predetermined *process identifier*, and this is denoted parametrically in the system by the special variable `id`. For presentation purpose, the concrete value of `id` for each individual process ranges from 1 to n. We note that the variable `id` cannot be changed. Even

though the processes are defined by one parameterized system, their dynamic behaviors can be arbitrarily different. This would depend on how `id` is expressed in the parameterized system.

Consider the 2-process parameterized system in Fig. 6.3(a), with (local) program points in angle brackets. Fig. 6.3(b) "concretizes" the processes explicitly. Note the use in the first process of the variable id_1 which is not writable in the process, and whose value is 1. Similarly, in the other process, id_2 has the value 2 and is not writable either.



(a) Parameterized System (b) Its 2-process Concretization

Figure 6.3: Example: Awaits then Increments

As before, a state $s \in \mathit{SymStates}$ comprises of two parts: the program point component and the symbolic constraint component. Now, however, the constraint component $[\![s]\!]$ also includes the *valuation* of the process identifiers. As such, $[\![s]\!]$ can now be denoted by a pair $\langle \mathtt{val}, \mathtt{pids} \rangle$, where `val` refers to the valuation of normal program variables while `pids` refers to the valuation of process identifiers. Note that all states from the same parameterized system share the same valuation of process identifiers. Therefore, when the context is clear, we omit the valuation `pids` of a state.

Again consider in Fig. 6.3 with two processes P_1 and P_2 with variables $\mathtt{id}_1 = 1$ and $\mathtt{id}_2 = 2$ respectively. In the system, it is specified parametrically that each process awaits for `x == id`. In P_1, this is interpreted as $\mathtt{await(x == id_1)}$ while P_2, this is interpreted as $\mathtt{await(x == id_2)}$. Each process has 2 transitions: the

first transfers it from control location $\langle 0\rangle$ to $\langle 1\rangle$, whereas the second transfers it from control location $\langle 1\rangle$ to $\langle 2\rangle$. Initially we have $x = 1$, i.e., the initial state s_0 is $\langle\langle 0,0\rangle, \langle x = 1, id_1 = 1 \wedge id_2 = 2\rangle\rangle$. We note that at s_0, both $t_1^{\{1\}}$ and $t_1^{\{2\}}$ are schedulable. However, among them, only $t_1^{\{1\}}$ is enabled. By taking transition $t_1^{\{1\}}$, P_1 moves from control location $\langle 0\rangle$ to $\langle 1\rangle$, and the whole system moves from state $\langle\langle 0,0\rangle, \langle x = 1, id_1 = 1 \wedge id_2 = 2\rangle\rangle$ to state $\langle\langle 1,0\rangle, \langle x = 1, id_1 = 1 \wedge id_2 = 2\rangle\rangle$. We note that here the transition $t_1^{\{2\}}$ is still disabled. From now on, let us omit the valuation of process identifiers. The whole system then takes the transition $t_2^{\{1\}}$ and moves from state $\langle\langle 1,0\rangle, x = 1\rangle$ to state $\langle\langle 2,0\rangle, x = 2\rangle$. Now, $t_1^{\{2\}}$ becomes enabled. Subsequently, the system takes $t_1^{\{2\}}$ and $t_2^{\{2\}}$ to move to state $\langle\langle 2,1\rangle, x = 1\rangle$ and finally to state $\langle\langle 2,2\rangle, x = 3\rangle$.

6.2.1 Symmetry

Given an n-process system, let $\mathcal{I} = [1 \cdots n]$ denote its *indices*, to be thought of as process identifiers. We write $Sym\,\mathcal{I}$ to denote the set of all permutations π on index set $\mathcal{I}$. Let Id be the identity permutation and π^{-1} the inverse of π.

For an indexed object b, such as a program point, a variable, a transition, valuation of program variables, or a formula, whose definition depends on $\mathcal{I}$, we can define the notion of permutation π acting on b, by simultaneously replacing each occurrence of index $i \in \mathcal{I}$ by $\pi(i)$ in b to get the result of $\pi(b)$.

EXAMPLE 6.3 : Consider the system in Fig. 6.3(b). Let the permutation π swap the two indices $(1 \mapsto 2, 2 \mapsto 1)$. Applying π to the valuation $x = 1$ gives us $\pi(x = 1) \equiv x = 1$, as x is a shared variable. Applying π to the formula $x = id_1 \wedge id_1 = 1$ gives us $\pi(x = id_1 \wedge id_1 = 1) \equiv (x = id_2 \wedge id_2 = 1)$. On the other hand, applying π to the transition $t_1^{\{1\}} \equiv \texttt{await(x = id}_1\texttt{)}$ will result in $\pi(t_1^{\{1\}}) \equiv t_1^{\{2\}} \equiv \texttt{await(x = id}_2\texttt{)}$.

Definition 32. *For $\pi \in Sym\,\mathcal{I}$ and state $s \in$ SymStates, $s \equiv \langle \ell, \langle \boldsymbol{val}, \boldsymbol{pids}\rangle\rangle$, the application of π on s is defined as $\langle \pi(\ell), \langle \pi(\boldsymbol{val}), \boldsymbol{pids}\rangle\rangle$.* □

In other words, permutations do not affect the valuation of process identifiers.

EXAMPLE 6.4 : Consider again the system in Fig. 6.3(b). Assume the π is the permutation swapping the 2 indices $(1 \mapsto 2, 2 \mapsto 1)$. We then have $\pi(\langle\langle 1{,}0\rangle, \langle x = 1, id_1 = 1 \wedge id_2 = 2\rangle\rangle) \equiv \langle\langle 0{,}1\rangle, \langle x = 1, id_1 = 1 \wedge id_2 = 2\rangle\rangle$. Please note that while π has no effect on shared variable x and valuation of process identifiers id_1, id_2, it does permute the local program points.

Definition 33. *For $\pi \in Sym\ \mathcal{I}$, a safety property ψ is said to be* symmetric *wrt. π if $\psi \equiv \pi(\psi)$.* □

We next present a traditional notion of symmetry.

Definition 34 (Strong Symmetry)**.** *For all $\pi \in Sym\ \mathcal{I}$, all safety property ψ, and all $s, s' \in \text{SymStates}$, we say that s is strongly π-similar to s' wrt. ψ, denoted by $s \overset{\pi,\psi}{\approx} s'$ if ψ is symmetric wrt. π and the following conditions hold:*

- $\pi(s) = s'$
- *for each transition t such that $s \xrightarrow{t} d$ we have $s' \xrightarrow{\pi(t)} d'$ and $d \overset{\pi,\psi}{\approx} d'$*
- *for each transition t' such that $s' \xrightarrow{t'} d'$ we have $s \xrightarrow{\pi^{-1}(t')} d$ and $d \overset{\pi,\psi}{\approx} d'$.* □

One of the strengths of this work is to allow symmetry by *disregarding* the values of the program variables.

Definition 35 (Weak Symmetry)**.** *For all $\pi \in Sym\ \mathcal{I}$, all safety property ψ, and all $s, s' \in \text{SymStates}$ such that $s \equiv \langle \ell, [\![s]\!]\rangle$ and $s' \equiv \langle \ell', [\![s']\!]\rangle$, we say that s is weakly π-similar to s' wrt. ψ, denoted by $s \overset{\pi,\psi}{\sim} s'$ if ψ is symmetric wrt. π and the following conditions hold:*

- $\pi(\ell) \equiv \ell'$
- $[\![s]\!] \models \psi$ `iff` $[\![s']\!] \models \pi(\psi)$
- *for each transition t such that $s \xrightarrow{t} d$ we have $s' \xrightarrow{\pi(t)} d'$ and $d \overset{\pi,\psi}{\sim} d'$*
- *for each transition t' such that $s' \xrightarrow{t'} d'$ we have $s \xrightarrow{\pi^{-1}(t')} d$ and $d \overset{\pi,\psi}{\sim} d'$.* □

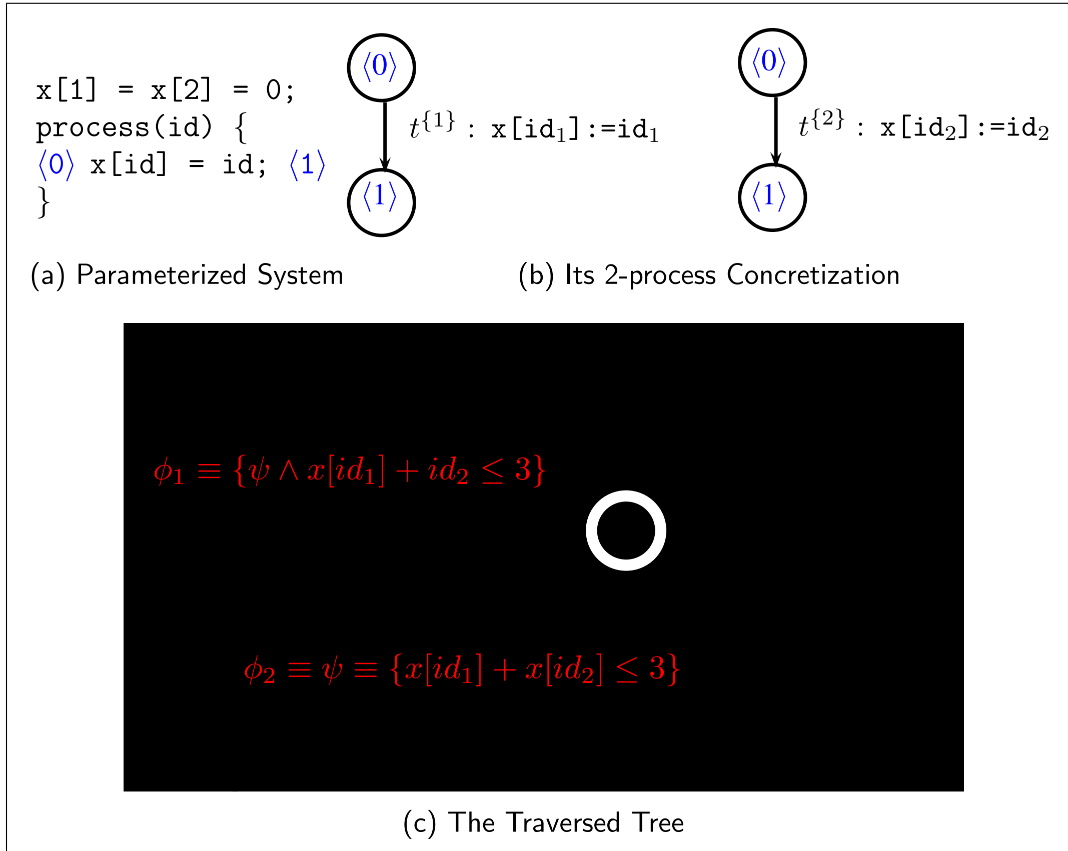


Figure 6.4: Example: Assign `id` to `x[id]`

We note here that, from now on, unless otherwise mentioned, symmetry means *weak* symmetry while π-similar means *weakly* π-similar. Also, it trivially follows that if s is π-similar to s' then s' is π^{-1}-similar to s. Consequently, if s is symmetric with s', then s' is symmetric with s too.

6.3 Motivating Examples

EXAMPLE 6.5 : Fig. 6.4 shows a parameterized system and its 2-process concretization. The shared array `x` contains 2 elements, initially 0. For convenience, we assume that array index starts from 1. Process 1 assigns $\texttt{id}_1$ (whose value is 1) to $\texttt{x}[1]$ while process 2 assigns $\texttt{id}_2$ (whose value is 2) to $\texttt{x}[2]$.

Consider the safety property $\psi \equiv x[1] + x[2] \leq 3$, interpreted as $\psi \equiv x[id_1] + x[id_2] \leq 3$. The reachability tree explored is in Fig. 6.4(c). Circles are used to denote states, while double-boundary circles denote subsumed/pruned states.
From the initial state $s_0 \equiv \langle\langle 0{,}0\rangle, \langle x[1] = 0 \wedge x[2] = 0, id_1 = 1 \wedge id_2 = 2\rangle\rangle$ process 1 progresses first and moves the system to the state:

$s_1 \equiv \langle\langle 1{,}0\rangle, \langle x[1] = 1 \wedge x[2] = 0, id_1 = 1 \wedge id_2 = 2\rangle\rangle$.

From s_1, process 2 now progresses and moves the system to the state:

$s_2 \equiv \langle\langle 1{,}1\rangle, \langle x[1] = 1 \wedge x[2] = 2, id_1 = 1 \wedge id_2 = 2\rangle\rangle$.

Note that s_0, s_1, and s_2 are all safe wrt. ψ. As there is no transition emanating from s_2, the interpolant for s_2 is computed as $\overline{\Psi}_2 \equiv \psi \equiv x[id_1] + x[id_2] \leq 3$. The pair $\langle\langle 1{,}1\rangle, \overline{\Psi}_2\rangle$ is memoized. The interpolant for s_1 can be computed as a conjunction of two formulas. One concerns the safety of s_1 itself, and the other concerns the safety of the successor state from $t^{\{2\}}$. In other words, we can have $\overline{\Psi}_1 \equiv \psi \wedge \mathsf{pre}(t^{\{2\}}, \psi)$, where $\mathsf{pre}(t, \phi)$ denotes a precondition wrt. to the program transition t and the postcondition ϕ. Consequently, we can have $\overline{\Psi}_1 \equiv \psi \wedge x[id_1] + id_2 \leq 3$. The pair $\langle\langle 1{,}0\rangle, \overline{\Psi}_1\rangle$ is memoized.

Now we arrive at state $s'_1 \equiv \langle\langle 0{,}1\rangle, \langle x[1] = 0 \wedge x[2] = 2, id_1 = 1 \wedge id_2 = 2\rangle\rangle$. This is indeed a symmetric image of state s_1 which we have explored and proved to be safe before. Here, we discover the permutation π to transform the program point $\langle 1{,}0\rangle$ to program point $\langle 0{,}1\rangle$. Clearly π simply swaps the two indices. We also observe that the safety property ψ is symmetric wrt. this π, i.e., $\pi(\psi) \equiv \psi$ (ψ is *invariant* wrt. π). In the next step, we check whether $[\![s'_1]\!]$ implies the *transformed interpolant* $\pi(\overline{\Psi}_1)$. We have $\pi(\overline{\Psi}_1) \equiv \pi(x[id_1] + x[id_2] \leq 3 \wedge x[id_1] + id_2 \leq 3)$ $\equiv x[id_2]+x[id_1] \leq 3 \wedge x[id_2]+id_1 \leq 3$. As $[\![s'_1]\!] \models x[id_2]+x[id_1] \leq 3 \wedge x[id_2]+id_1 \leq 3$, we do not need to explore s'_1 any further. In other words, the subtree rooted at s'_1 is pruned.

EXAMPLE 6.6 : Consider the concurrent system in Fig. 6.5. We are interested in safety property $\psi \equiv x < 2$. As x is a shared variable, ψ is symmetric wrt. all

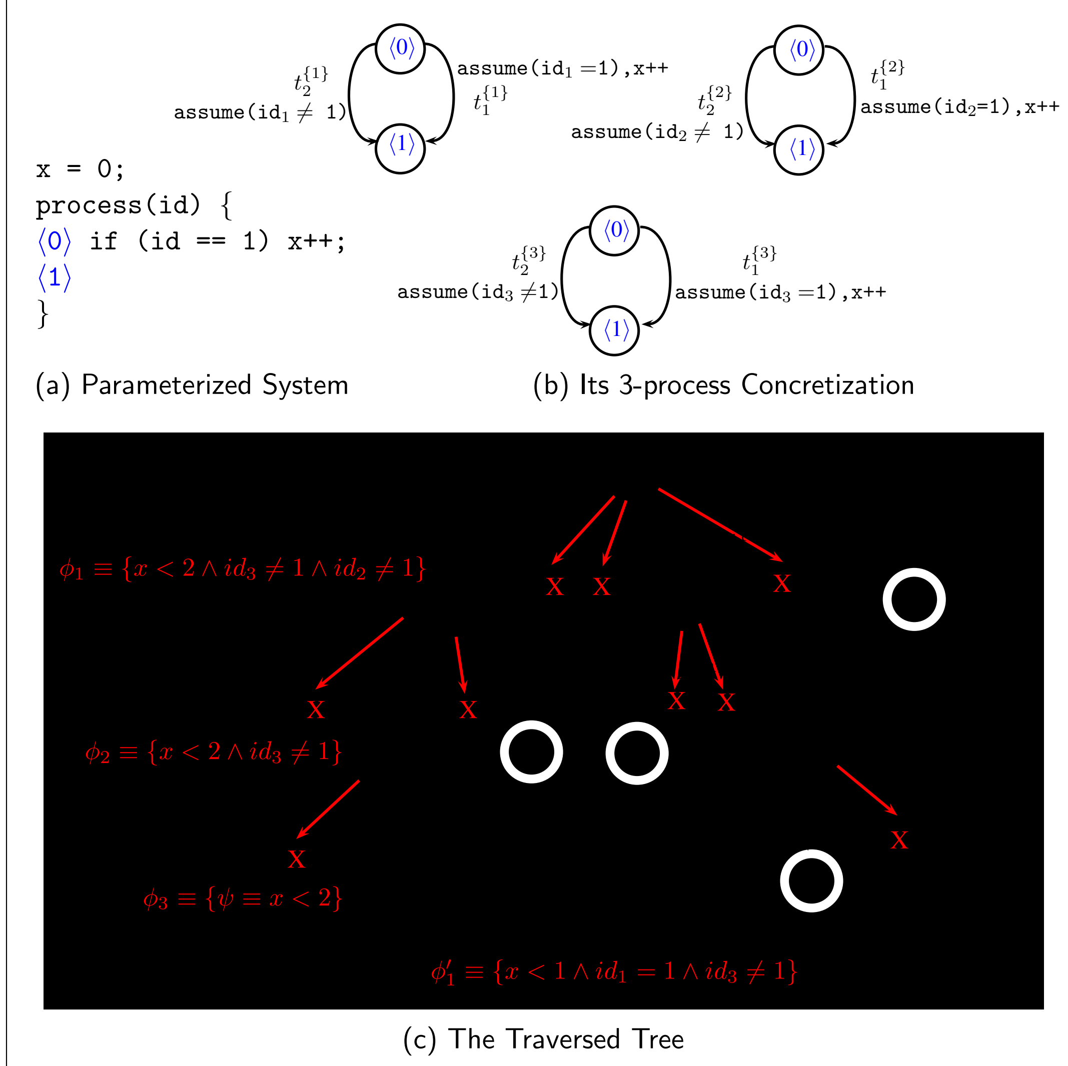


Figure 6.5: Example: Only Process #1 Increments

possible permutations.

The reachability tree is depicted in Fig. 6.5(c). From the initials state s_0 we arrive at states s_1, s_2, and s_3, where:

$s_0 \equiv \langle\langle 0{,}0{,}0\rangle, \langle x = 0, id_1 = 1 \wedge id_2 = 2 \wedge id_3 = 3\rangle\rangle$

$s_1 \equiv \langle\langle 1{,}0{,}0\rangle, \langle x = 1, id_1 = 1 \wedge id_2 = 2 \wedge id_3 = 3\rangle\rangle$

$s_2 \equiv \langle\langle 1{,}1{,}0\rangle, \langle x = 1, id_1 = 1 \wedge id_2 = 2 \wedge id_3 = 3\rangle\rangle$

$s_3 \equiv \langle\langle 1{,}1{,}1\rangle, \langle x = 1, id_1 = 1 \wedge id_2 = 2 \wedge id_3 = 3\rangle\rangle.$

At s_3 we compute its interpolant $\overline{\Psi}_3 \equiv \psi \equiv x < 2$. In a similar manner as before, we compute the interpolant for s_2, which is $\overline{\Psi}_2 \equiv x < 2 \wedge id_3 \neq 1$. When we are at state $s'_2 \equiv \langle\langle 1{,}0{,}1\rangle, \langle x = 1, id_1 = 1 \wedge id_2 = 2 \wedge id_3 = 3\rangle\rangle$, we look for a permutation π_1 such that $\pi_1(\langle 1{,}1{,}0\rangle) \equiv \langle 1{,}0{,}1\rangle$. Clearly we can have π_1 as the permutation which fixes the first index and swaps the last 2 indices. Moreover, $[\![s'_2]\!] \equiv \langle x = 1, id_1 = 1 \wedge id_2 = 2 \wedge id_3 = 3\rangle \models \pi_1(\overline{\Psi}_2) \equiv x < 2 \wedge id_2 \neq 1$. Therefore, s'_2 is pruned.

Similarly, the interpolant $\overline{\Psi}_1$ for s_1 is computed as $x < 2 \wedge id_2 \neq 1 \wedge id_3 \neq 1$. When at state $s'_1 \equiv \langle\langle 0{,}1{,}0\rangle, \langle x = 0, id_1 = 1 \wedge id_2 = 2 \wedge id_3 = 3\rangle\rangle$, we look for a permutation π_2 such that $\pi_2(\langle 1{,}0{,}0\rangle) \equiv \langle 0{,}1{,}0\rangle$. Clearly we can have π_2 as the permutation which fixes the third index and swaps the first two indices. However, $[\![s'_1]\!] \equiv \langle x = 0, id_1 = 1 \wedge id_2 = 2 \wedge id_3 = 3\rangle \not\models \pi_2(\overline{\Psi}_1) \equiv x < 2 \wedge id_1 \neq 1 \wedge id_3 \neq 1$. Thus the subtree rooted at s'_1 cannot be pruned and it requires further exploration. After s'_1 has been traversed, the interpolant for s'_1 is computed as $\overline{\Psi}'_1 \equiv x < 1 \wedge id_1 = 1 \wedge id_3 \neq 1$. Next we visit $s''_1 \equiv \langle\langle 0{,}0{,}1\rangle, \langle x = 0, id_1 = 1 \wedge id_2 = 2 \wedge id_3 = 3\rangle\rangle$. We can find a permutation π_3 which fixes the first index and swaps the last 2 indices ($\pi_3 \equiv \pi_1$). We have $\pi_3(\langle 0{,}1{,}0\rangle \equiv \langle 0{,}0{,}1\rangle$. Also $[\![s''_1]\!] \equiv \langle x = 0, id_1 = 1 \wedge id_2 = 2 \wedge id_3 = 3\rangle \models \pi_3(\overline{\Psi}'_1) \equiv x < 1 \wedge id_1 = 1 \wedge id_2 \neq 1$. As a result, we can avoid considering the subtree rooted at s''_1.

In the two above examples, we have shown how the concept of interpolation can help capture the shape of a subtree. More importantly, computed interpolants can be transformed in order to detect the symmetry as well as the non-symmetry (mainly due to the use of `id`) between candidate subtrees.

6.4 Complete Symmetry Reduction Algorithm

Our algorithm, presented in Fig. 6.6, naturally performs a depth first search of the interleaving tree. It assumes the safety property to be known as ψ. Initially, we explore the initial state s_0 with an empty *history* h. During the search process,

Assume safety property ψ and initial state s_0
⟨1⟩ *Initially* : Explore$(s_0, \emptyset)$
function Explore(s, h)
Let s be $\langle \ell, [\![s]\!] \rangle$
⟨2⟩ **if** $[\![s]\!] \not\models \psi$ `Report Error and TERMINATE`
⟨3⟩ **if** $\exists\, \pi \bullet \pi(\psi) \equiv \psi \wedge \exists\, \ell' \bullet \ell \equiv \pi(\ell') \wedge \exists\, \overline{\Psi} \bullet \mathsf{memoed}(\ell', \overline{\Psi}) \wedge [\![s]\!] \models \pi(\overline{\Psi})$
return $\pi(\overline{\Psi})$
⟨4⟩ **if** $s \in h$ /* We hit a cycle */
⟨5⟩ Let θ be the cyclic path
⟨6⟩ Assert$(Cyclic(s, \theta))$
⟨7⟩ **return** `true` /* Initial value for fix-point computation */
else
⟨8⟩ $h := h \cup \{s\}$
endif
⟨9⟩ $\overline{\Psi} := \psi$
⟨10⟩ **foreach** $t \in Schedulable(s)$ **do**
⟨11⟩ **if** $t \in Enabled(s)$
⟨12⟩ $s \xrightarrow{t} s'$ /* Execute t */
⟨13⟩ $\overline{\Psi}' :=$ Explore(s', h)
⟨14⟩ $\overline{\Psi} := \overline{\Psi} \wedge \mathsf{pre}(t, \overline{\Psi}')$
else
⟨15⟩ $\overline{\Psi} := \overline{\Psi} \wedge \mathsf{pre}(t, false)$
endif
⟨16⟩ **endfor**
⟨17⟩ Let Θ *be* $\{\theta \mid Cyclic(s, \theta)\}$
⟨18⟩ **if** $(\Theta \neq \emptyset)$ $\overline{\Psi} :=$ FIX-POINT$(s, \Theta, \overline{\Psi})$
/* s is a loop point, so we ensure $\overline{\Psi}$ is an invariant along the paths Θ */
⟨19⟩ Retractall$(Cyclic(s, \theta))$
⟨20⟩ $h := h \setminus \{s\}$
⟨21⟩ memo$(\ell, \overline{\Psi})$ and **return** $\overline{\Psi}$
end function

Figure 6.6: Complete Symmetry Reduction Algorithm (DFS)

the function Explore will be recursively called. Note that termination is achieved by assuming finite setting.

Base Cases: The first base case is when the current state does not conform to the safety property ψ (line 2). We then immediately report an error and terminate. The second base case applies when the current state (subtree) has a symmetric image

(subtree) which has already been traversed and proved to be safe before (line 3). We have well exemplified such scenarios in previous Sections.

The third base case requires some elaboration. Using the history h, we detect a cycle (line 4). Specifically, there is a cyclic path θ from s back to s. We note this down and return true. Later, after the descendants of s have been traversed, we require a fix-point computation of the interpolant for s, as shown in line 17-18. The function FIX-POINT computes an invariant interpolant for s, wrt. the initial value $\overline{\Psi}$ and the set of cyclic paths Θ. Essentially, this function involves computing, for each cyclic path, a *path invariant*. Such a computation is performed backwards, using a previously computed invariant at the bottom of the cyclic path, and then extracting a new invariant for s. Then each computed path invariant is fed into other paths in order to compute a new invariant. The process terminates at a fix-point. Assuming finite setting, termination is then guaranteed because of monotonicity of the path invariant computation and the fact that there are only finitely many possible invariants (note $[\![s]\!]$ itself is an invariant). Finally, the interpolant for each state appearing in these cyclic paths are now updated appropriately. This is in light of now having an invariant for all of them simultaneously.

We remark here that this fix-point task, though seemingly complicated, is in fact routine. We refer interested readers to [Jaffar *et al.*, 2011] for more details regarding this matter. We also remark that for many concurrent protocols, where involved operations are mainly "set" and "re-set" operations, a fix-point is achieved just after one iteration.

Recursive Traversal and Computing the Interpolants: Our algorithm recursively explores the successors of the current state by the recursive call in line 13. The interpolant $\overline{\Psi}$ for the current state is computed as from line 9-18. As mentioned above, cyclic paths are handled in line 17-18. The operation $\mathsf{pre}(t, \phi)$ denotes the precondition computation wrt. the program transition t and the postcondition ϕ. In practice, we implement this as an approximation of the weakest precondition

computation [Dijkstra, 1975], as in previous Chapters.

Theorem 5 (Soundness)**.** *Our symmetry reduction algorithm is sound.*

Here, by soundness, we mean that all pruning performed in line 3 will not affect the verification result.

Proof Outline. *Let the triple* $\{\overline{\Psi}\}\ \langle\langle pc_1, pc_2, \cdots, pc_n\rangle; P_1||P_2||\cdots||P_n\rangle\ \{\psi\}$ *denote the fact that* $\overline{\Psi}$ *is a* sound *interpolant for program point* $\langle pc_1, pc_2, \cdots, pc_n\rangle$ *wrt. the safety property* ψ *and the concurrent system* $P_1||P_2||\cdots||P_n$*. We will not prove that our interpolant computation (line 9-18) is a sound computation. Instead, we assume such soundness by following [Jaffar et al., 2009; Jaffar et al., 2011]. Let us assume that the soundness of that triple is witnessed by a proof* $\mathcal{P}$*. By consistently renaming* $\mathcal{P}$ *with a renaming function* $\pi \in Sym\ \mathcal{I}$*, we can derive a new* sound *fact (i.e., a proof), which is:*

$\{\pi(\overline{\Psi})\}\ \pi(\langle\langle pc_1, pc_2, \cdots, pc_n\rangle; P_1||P_2||\cdots||P_n\rangle)\ \{\pi(\psi)\} \equiv$

$\{\pi(\overline{\Psi})\}\ \langle\langle pc_{\pi(1)}, pc_{\pi(2)}, \cdots, pc_{\pi(n)}\rangle; P_{\pi(1)}||P_{\pi(2)}||\cdots||P_{\pi(n)}\rangle\ \{\pi(\psi)\}$

Since $P_1, P_2, \cdots P_n$ *come from the same parameterized system and* π *is a bijection on* $\mathcal{I}$*, we have:*

$P_{\pi(1)}||P_{\pi(2)}||\cdots||P_{\pi(n)} \equiv P_1||P_2||\cdots||P_n$

Therefore, $\{\pi(\overline{\Psi})\}\ \langle\langle pc_{\pi(1)}, pc_{\pi(2)}, \cdots, pc_{\pi(n)}\rangle; P_1||P_2||\cdots||P_n\rangle\ \{\pi(\psi)\}$ *must hold too. In the case that* ψ *is symmetric wrt.* π*, we have* $\pi(\psi) \equiv \psi$*. Thus* $\pi(\overline{\Psi})$ *is a* sound *interpolant for program point* $\langle pc_{\pi(1)}, pc_{\pi(2)}, \cdots, pc_{\pi(n)}\rangle$ *wrt. the same safety property* ψ *and the same concurrent system* $P_1||P_2||\cdots||P_n$*. As a result, the use of interpolant* $\pi(\overline{\Psi})$ *for pruning (at line 3 in Fig. 6.6) is* sound. □

Definition 36 (Symmetry Preserving Precondition Computation)**.** *Given a parametrically defined n-process system and a safety property* ψ*, the precondition computation* **pre** *used in our algorithm is said to be* symmetry preserving *if for all* $\pi \in Sym\ \mathcal{I}$*, for all transition* t *and all postcondition* ϕ • $\pi(\mathsf{pre}(t, \phi)) \equiv \mathsf{pre}(\pi(t), \pi(\phi))$*.*

□

This property means that our precondition computation is consistent wrt. renaming operation. In other words, the implementation of pre is *independent* of the naming of variables contained in its inputs. A reasonable implementation of pre can easily ensure this.

Definition 37 (Monotonic Precondition Computation)**.** *Given a parametrically defined n-process system and a safety property ψ, the precondition computation pre used in our algorithm is said to be* monotonic *if for all transition t and all postconditions $\phi_1, \phi_2 \bullet \phi_1 \rightarrow \phi_2$ implies $\mathit{pre}(t, \phi_1) \rightarrow \mathit{pre}(t, \phi_2)$.* □

We emphasize here that the weakest precondition computation [Dijkstra, 1975] does possess the monotonicity property. As is well-known, computing the weakest precondition in all the cases is very expensive. However, in practice (and in particular in the experiments we have performed), we often observe this monotonicity property with the implementation of our precondition computation. Possible implementations for this operation are discussed in Chapter 3 and also in [Rybalchenko and Sofronie-Stokkermans, 2007; Jaffar *et al.*, 2009; Jaffar *et al.*, 2011].

Definition 38 (Completeness in Symmetry Reduction)**.** *In proving a parametrically defined n-process system with a global state space* SymStates *and a safety property ψ, an algorithm which traverses the reachability tree is said to be* complete *wrt. a symmetry relation $\mathcal{R}$* `iff` *for all $s, s' \in$ SymStates, $s \; \mathcal{R} \; s'$ implies that the algorithm will avoid traversing either the subtree rooted at s or the subtree rooted at s'.* □

We remark here that our definition of completeness does not concern with the power of an algorithm in giving the answer to a safety verification question. This definition of completeness, however, is about the power of an algorithm in exploiting symmetry for search space reduction.

Theorem 6 (Completeness)**.** *Our symmetry reduction algorithm is complete wrt. the weak symmetry relation if our operation pre is both monotonic and symmetry preserving.*

Proof Outline. *Assume that* $s, s' \in \mathrm{SymStates}$ *and* s *is weakly* π*-similar to* s'*. W.l.o.g. assume we encounter* s *first. If the subtree rooted at* s *is pruned (due to subsumption), the theorem trivially holds. The theorem also trivially holds if* s *is* not *a safe root. Now we consider that the subtree rooted as* s *is proved to be safe and the returned interpolant is* $\overline{\Psi}$*. We will prove by structural induction on this interpolated subtree that* s' *will indeed be pruned, i.e.,* $[\![s']\!] \models \pi(\overline{\Psi})$*.*

For simplicity of the proof, we will prove for loop-free programs only. In other words, we ignore our loop handling mechanism (line 4-7,17-18). Note that our theorem still holds for the general case. However, to prove this, we will require another induction on our fix-point computation in line 18.

For the base case that $\overline{\Psi}$ *is* ψ *(when there is no schedulable transition from* s*) due to the definition of weak symmetry relation, there is no schedulable transition from* s' *and* $[\![s']\!] \models \pi(\psi)$*. Therefore, traversing the subtree rooted at* s' *is avoided.*

As the induction hypothesis, assume now that the theorem holds for all the descendants of state s*. Let assume that* $\overline{\Psi} \equiv \psi \wedge \overline{\Psi}_1 \wedge \overline{\Psi}_2 \wedge \cdots \wedge \overline{\Psi}_k \wedge \overline{\Psi}_{k+1} \wedge \cdots \wedge \overline{\Psi}_m$*, where* $\overline{\Psi}_1 \cdots \overline{\Psi}_k$ *are the interpolants contributed by enabled transitions in* s *and* $\overline{\Psi}_{k+1} \cdots \overline{\Psi}_m$ *are the interpolants contributed by schedulable but disabled transitions in* s *(line 14 and 15). Now assume the contrary that* $[\![s']\!] \not\models \pi(\overline{\Psi})$*. We will show that this would lead to a contradiction. Using the first condition of weak symmetry relation, obviously* $[\![s']\!] \models \pi(\psi)$*. As such, there must exist some* $1 \leq j \leq m$ *such that* $[\![s']\!] \not\models \pi(\overline{\Psi}_j)$*. There are two possible cases: (1)* $\overline{\Psi}_j$ *is contributed by an enabled transition; (2)* $\overline{\Psi}_j$ *is contributed by a disabled, but schedulable transition.*

Let us consider case (1) first. Assume $\overline{\Psi}_j$ *corresponds to transition* $t \in Enabled(s)$ *and* $s \overset{t}{\rightarrow} d$*. By definition we have* $s' \overset{\pi(t)}{\rightarrow} d'$ *and* d *is weakly* π*-similar to* d'*. Let* $\overline{\Psi}_d$ *be interpolant for the subtree rooted at* d*. By induction hypothesis, we have* $[\![d']\!] \models \pi(\overline{\Psi}_d)$*. Obviously, we have* $[\![s']\!] \models \mathsf{pre}(\pi(t), [\![d']\!])$*, by monotonicity of* pre*, we deduce* $[\![s']\!] \models \mathsf{pre}(\pi(t), \pi(\overline{\Psi}_d))$*. As* pre *is symmetry preserving,* $[\![s']\!] \models \mathsf{pre}(\pi(t), \pi(\overline{\Psi}_d)) \equiv \pi(\mathsf{pre}(t, \overline{\Psi}_d)) \equiv \pi(\overline{\Psi}_j)$*. Consequently we arrive at the fact that*

$[\![s']\!] \models \pi(\overline{\Psi}_j)$ *which is a contradiction.*

For case (2), by using the symmetry preserving property of pre *and the fact that* $\pi(false) \equiv false$*, we also derive a contradiction.* □

6.5 Experimental Evaluation

We used a 3.2 GHz Intel processor and 2GB memory running Linux. Unless otherwise mentioned, timeout is set at 300 seconds, and '-' indicates timeout. In this section, we benchmark our proposed approach, namely Complete Symmetry Reduction (CSR), against current state-of-the-arts.

	CSR			RSR			NSR		
# Phil	Visited	Subsumed	T(s)	Visited	Subsumed	T(s)	Visited	Subsumed	T(s)
3	68	29	0.02	67	27	0.02	191	79	0.06
4	230	134	0.09	328	184	0.13	1246	702	0.81
5	662	446	0.28	1509	981	0.71	7517	4893	4.93
6	1778	1304	0.85	7356	5216	4.18	43580	30908	34.53
7	4584	3552	2.55	35079	26335	28.83	–	–	–
8	11526	9281	7.54	–	–	–	–	–	–
9	28287	23432	22.6	–	–	–	–	–	–
10	67920	57504	58.07	–	–	–	–	–	–
11	159738	137609	226.86	–	–	–	–	–	–

Table 6.1: Experiments on Dining Philosophers

Our first example is the classic *dining philosophers* problem. As commonly known, it exhibits *rotational* symmetry. However, and more importantly, we exploit far more symmetry than that. In details, at *any* program point, rotational symmetry is applicable. Nevertheless, for certain program points, when some transitions have been taken, the system exhibits more symmetry than just rotational symmetry. With this benchmark, we demonstrate the power of our complete symmetry reduction (CSR) algorithm. Here, we verify a *tight* safety property that "no more than *half* the philosophers can eat simultaneously".

Table 6.1 presents three variants: Complete Symmetry Reduction (CSR), Rota-

tional Symmetry Reduction (RSR), and No Symmetry Reduction (NSR). The number of *stored states* is the difference between the number of visited states (Visited column) and subsumed states (Subsumed column). Note that although RSR achieves linear reduction compared to NSR, it does not scale well. CSR significantly outperforms RSR and NSR in all the instances.

		Complete Symmetry Reduction			Lazy Symmetry Reduction	
# Readers	# Writers	Visited	Subsumed	T(s)	Abstract States	T(s)
2	1	35	20	0.01	9	0.01
4	2	226	175	0.19	41	0.10
6	3	779	658	0.93	79	67.80
8	4	1987	1750	3.23	165	81969.00
10	5	4231	3820	9.21	–	–

Table 6.2: Experiments on Reader-Writer Protocol

Next consider the *Reader-Writer Protocol* from [Wahl, 2007; Wahl and D'Silva, 2010]. Here we highlight the aspect of *search space size* as compared to top-down techniques, of which the most recent implementation of Lazy Symmetry Reduction [Wahl and D'Silva, 2010] is chosen as a representative [1]. Table 6.2 shows that although lazy symmetry reduction has aggressively compressed the state space (which now grows roughly in linear complexity), the running time is still *exponential*. In other words, the number of abstract states is not representative of the search space. In contrast, the running time of our method is significantly better. In the instance of 8 readers and 4 writers, we extended the timeout for [Wahl and D'Silva, 2010] to finish; and it takes almost 1 day.

Next we experiment with the "Sum-of-ids" example mentioned earlier. To the best of our knowledge, there is no symmetry reduction algorithm which can detect and exploit symmetry here. Table 6.3 shows we have significant symmetry reduction. In term of memory (stored states), we enjoy linear complexity. For reference, we also report the running time of this example, without symmetry reduction, using SPIN 5.1.4 [SPIN,].

[1] We receive this implementation from the authors of [Wahl and D'Silva, 2010].

	Complete Symmetry Reduction			SPIN-NSR		
# Processes	Visited	Subsumed	T(s)	Visited	Subsumed	T(s)
10	57	45	0.02	6146	4097	0.03
20	212	190	0.04	11534338	9437185	69.70
40	822	780	0.37	–	–	–
60	1832	1770	1.91	–	–	–
80	3242	3160	7.62	–	–	–
100	5052	4950	22.09	–	–	–

Table 6.3: Experiments on Sum-of-ids Example

	Complete Symmetry Reduction			SI		
# Processes	Visited	Subsumed	T(s)	Visited	Subsumed	T(s)
3	65	31	0.10	265	125	0.43
4	182	105	0.46	1925	1089	5.89
5	505	325	2.26	14236	9067	74.92
6	1423	983	11.10	–	–	–

Table 6.4: Experiments on Bakery Algorithm

In the fourth and last example, we apply our method to handle infinite domain variables and loops. We choose the well-known Bakery algorithm to perform the experiments, and we use the well-known abstraction of using an inequality to describe each pair of counters to close the loops. Again, as far as we are aware of, there has been no symmetry reduction algorithm which can detect and exploit symmetry for this example. Table 6.4 shows the significant improvements due to our symmetry reduction, compared to just symbolic execution with interpolation, denoted as SI.

6.6 Summary

We presented a method of symmetry reduction for searching the interleaving space of a concurrent system of transitions in pursuit of a safety property. The class of systems considered, by virtue of being defined parametrically, is completely general; the individual processes may be at any level of similarity to each other. We then enhanced a general method of symbolic execution with interpolation for traditional safety verification of transition systems, in order to deal with symmetric states.

We then defined a notion of weak symmetry, one that allows for more symmetry than the stronger notion that is used in the literature. Finally, we showed that our method, when employed with an interpolation algorithm which is monotonic, can exploit weak symmetry completely.

Chapter 7

Conclusion

The true function of philosophy is to educate us in the principles of reasoning and not to put an end to further reasoning by the introduction of fixed conclusions.

George Henry Lewes

This Chapter concludes the thesis. We will summarize the thesis in Section 7.1 and informally discuss some foreseeable impacts of this thesis in Section 7.2.

7.1 Summary

In this thesis, we study the framework for program reasoning founded upon symbolic execution. As discussed, symbolic execution is intuitive while very powerful since it enables us to potentially obtain fully accurate reasoning. We apply this reasoning framework to two important and extremely difficult domain areas, namely *program path analysis* and *safety verification of concurrent programs*. The main challenge comes from the path explosion problem of symbolic execution, due to the extremely high demand for path-sensitivity, which by nature is inevitable in the domain areas. This thesis contributes by proposing custom interpolation methods, target for the

two domain areas, and specifically address the scalability issues caused by loops and interleavings. We again briefly summarize our contributions as below.

First, we address the Worst-Case Execution Time (WCET) *path analysis* problem for bounded programs, formalized as discovering a tight upper bound of a timing variable. A key challenge is posed by complicated loops whose iterations exhibit non-uniform behavior. Traditional methods such as abstract interpretation often are too inaccurate. We adopt a brute-force strategy by simply *unrolling* loops, and show how to make this scalable while preserving accuracy.

Our algorithm performs *symbolic simulation* of the program. It maintains accuracy because it preserves, at critical points, *path-sensitivity*. In other words, the simulation detects infeasible paths. Scalability, on the other hand, is dealt with by using *summarizations*, compact representations of the analyses of loop iterations. They are obtained by a judicious use of abstraction which preserves critical information flowing from one iteration to another. These summarizations can be *compounded* in order for the simulation to have *linear complexity*: the symbolic execution can in fact be *asymptotically shorter* than a concrete execution. This is important because the cost of symbolic simulation is, clearly, far higher than concrete simulation.

Second, we consider the path analysis problem for general resource usage. This includes the analysis of non-cumulative resource such as memory high watermark. For precision and practicality, we target our framework to accommodate both path sensitivity and user assertions. We show that, under a greedy treatment for loop to make the analysis scalable, enforcing assertions produces unsound results.

We address the challenge using a novel two-phase algorithm. The first phase employs a greedy strategy in the unrolling of loops. This unrolling explores and summarizes what is conceptually a symbolic execution tree, which is of enormous size. At the end of the first phase, we produce a compact representation by restricting attention only to the assertion variables. The simplified tree is represented

in the form of transitions in order to avoid an upfront consideration, which still remains exponential in the loop iterations. Finally, our second phase attacks the remaining problem, to determine the longest path in this simplified tree, directly with an adaptation of a dynamic programming algorithm.

Third, we consider the problem of reasoning about interleavings in safety verification of concurrent processes. We start with a systematic search tree depicting the program states across all possible interleavings. While this setting is totally general, a naive implementation based on explicit enumeration is clearly impractical. We then consider an algorithm schema which can prune the search space. We contribute by enhancing *trace-based* methods, collectively known as "Partial Order Reduction". Here we further weaken the concept of Partial Order Reduction to *Property Driven Partial Order Reduction* (PDPOR) — which is now *property dependent* — in order to adapt it for a symbolic execution framework with abstraction. Our main contribution, however, is a framework that synergistically combines state interpolation and PDPOR so that the sum is more than its parts.

Finally, we consider reduction technique for interleavings, but now under the assumption that processes are similarly defined via a parameterized system. The most prominent concept for this purpose is *symmetry reduction.* We define a notion of *weak* symmetry which is property dependent and allows for more symmetry than the stronger notion used in the literature. Our method, when employed with an interpolation algorithm which is monotonic, can exploit weak symmetry *completely.*

7.2 Concluding Remarks and Future Research

Our loop unrolling technique has overcome the fundamental problem of simulation techniques: the "depth" issue. Now symbolic execution can in fact be *asymptotically shorter* than a concrete execution. This is extremely important because the cost of symbolic simulation is, clearly, far higher than concrete simulation. The impact of this result is huge, as loop unrolling is commonly performed, either partially or

fully, in a wide range of analyses.

The marriage of a greedy treatment for loop with user assertions is thought provoking. Going greedy, we abstract away certain information, some of which might be quite relevant. Only by doing this we ensure scalability. However, to exploit user information, given as assertions, we need to narrow or zoom into a certain number of program paths. These two processes are fundamentally in conflict with each other. Investigating this fundamental conflict in a more general setting is left as future work.

Before this thesis, POR and symmetry reduction are investigated from the forward learning (or top-down) point of view. Usually, we investigate the program syntax in order to identify similarities which will arise in the search process. This static learning step is relatively cheap while significant pruning can be obtained. However, such learning phase is not sensitive wrt. the target safety property. On the other hand, in this thesis, our learning (of the interpolants) is dynamic, backward, and relative wrt. the given target. Of course, it gives rise to significantly more pruning, but at a non-trivial cost of more complicated algorithms. We believe that in many cases, these two learning paradigms can be very much compatible. Our work in combining state interpolation PDPOR, where PDPOR can well be reduced to traditional POR, somewhat suggests that compatibility. Exploring this direction is definitely an interesting future work.

Bibliography

[aiT,] aiT Worst-Case Execution Time Analyzers. URL `http://www.absint.com-/ait/index.htm`.

[Albarghouthi *et al.*, 2010] A. Albarghouthi, A. Gurfinkel, O. Wei, and M Chechik. Abstract analysis of symbolic executions. In *CAV*, 2010.

[Altenbernd, 1996] P. Altenbernd. On the false path problem in hard real-time programs. In *In Proceedings of the 8th Euromicro Workshop on Real-time Systems*, pages 102–107, 1996.

[Ball *et al.*, 2001] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*, 2001.

[Ball *et al.*, 2004] T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In *IFM*, 2004.

[Beckert *et al.*, 2007] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. 2007.

[Beyer *et al.*, 2007] D. Beyer, T.A. Henzinger, R. Jhala, and R. Majumdar. The Software Model Checker BLAST. *Int. J. STTT*, 9:505–525, 2007.

[Beyer *et al.*, 2009] D. Beyer, A. Cimatti, A. Griggio, M.E. Keremoglu, and R. Sebastiani. Software model checking via large-block encoding. In *FMCAD*, 2009.

[Bjørner *et al.*, 1997] N. Bjørner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *TCS*, 173(1):49–87, February 1997.

[Bokor *et al.*, 2011] P. Bokor, J. Kinder, M. Serafini, and N. Suri. Supporting domain-specific state space reductions through local partial-order reduction. In *ASE*, 2011.

[Bound-T,] Bound-T time and stack analyser. URL `http://www.bound-t.com`.

[Brooks Jr., 1995] F. P. Brooks Jr. *The mythical man-month (anniversary ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Bygde *et al.*, 2009] S. Bygde, A. Ermedahl, and B. Lisper. An efficient algorithm for parametric WCET calculation. In *RTCSA*, 2009.

[Cadar *et al.*, 2006] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically Generating Inputs of Death. In *CCS*, 2006.

[Cadar *et al.*, 2011] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: preliminary assessment. In *ICSE*, 2011.

[Chu and Jaffar,] D. H. Chu and J. Jaffar. A Framework for Combining State Interpolatin and Partil Order Reduction, journal = Under Submission, year = 2012.

[Chu and Jaffar, 2011] D. H. Chu and J. Jaffar. Symbolic simulation on complicated loops for wcet path analysis. In *EMSOFT*, 2011.

[Chu and Jaffar, 2012a] D. H. Chu and J. Jaffar. A Complete Method for Symmetry Reduction in Safety Verification. In *CAV*, 2012.

[Chu and Jaffar, 2012b] D. H. Chu and J. Jaffar. Path-Sensitive Resource Analysis Compliant with Assertions. *Under Submission*, 2012.

[Clarke *et al.*, 1993] E. M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In *CAV*, 1993.

[Clarke *et al.*, 1999] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[Clarke *et al.*, 2000] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. CounterrExample-Guided Abstraction Refinement. In *CAV*, 2000.

[Clarke, 1976] Lori A. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Trans. Software Eng.*, 1976.

[Collatz, 1937] Collatz. On the 3x+1 problem. Available at http://www.ericr.nl/wondrous, 1937.

[Cordeiro and Fischer, 2011] L. Cordeiro and B. Fischer. Verifying multi-threaded software using smt-based context-bounded model checking. In *ICSE*, 2011.

[Cousot and Cousot, 1977] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis. In *POPL*, 1977.

[Cousot and Halbwachs, 1978] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *POPL*, 1978.

[Craig, 1955] W. Craig. Three uses of Herbrand-Gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Computation*, 22, 1955.

[Dijkstra, 1972] E. W. Dijkstra. The humble programmer. *Commun. ACM*, 1972.

[Dijkstra, 1975] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 1975.

[Dillig *et al.*, 2008] Isil Dillig, Thomas Dillig, and Alex Aiken. Sound, complete and scalable path-sensitive analysis. In *PLDI*, 2008.

[Emerson and Sistla, 1993] E. A. Emerson and A. P. Sistla. Model checking and symmetry. In *CAV*, 1993.

[Emerson and Sistla, 1997] E. A. Emerson and A. P. Sistla. Utilizing symmetry when model-checking under fairness assumptions. *ACM TOPLAS*, 19(4):617–638, July 1997.

[Emerson and Trefler, 1999] E. A. Emerson and R. J. Trefler. From asymmetry to full symmetry: New techniques for symmetry reduction in model checking. In *Conference on Correct Hardware Design and Verification Methods*, 1999.

[Emerson *et al.*, 2000] E. A. Emerson, J. W. Havlicek, and R. J. Trefler. Virtual symmetry reduction. In *LICS*, 2000.

[Engblom and Ermedahl, 2000] J. Engblom and A. Ermedahl. Modeling complex flows for worst-case execution time analysis. *Real-Time Systems Symposium*, 2000.

[Ermedahl and Gustafsson, 1997] A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. In *Euro-Par*, 1997.

[Ermedahl *et al.*, 2003] A. Ermedahl, F. Stappert, and J. Engblom. Clustered calculation of worst-case execution times. In *CASES '03*, pages 51–62. ACM, 2003.

[Esteban and Genaim, 2012] D. Esteban and S. Genaim. On the limits of the classical approach to cost analysis. In *SAS*, 2012.

[Flanagan and Godefroid, 2005] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, 2005.

[Floyd, 1967] R. W. Floyd. Assigning meaning to programs. In *Proceedings of the Symposium on Applied Maths*, 1967.

[Garfinkel, 2005] S. Garfinkel. History's worst software bugs. URL http://www.wired.com/software/coolapps/news/2005/11/69355?currentPage=all, 2005.

[Godefroid, 1996] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem.* Springer-Verlag New York, Inc., 1996.

[Grumberg *et al.*, 2005] O. Grumberg, F. Lerda, O. Strichman, and M. Theobald. Proof-guided underapproximation-widening for multi-process systems. In *POPL*, 2005.

[Gueta *et al.*, 2007] G. Gueta, C. Flanagan, E. Yahav, and M. Sagiv. Cartesian partial-order reduction. In *SPIN*, 2007.

[Gulwani and Zuleger, 2010] S. Gulwani and F. Zuleger. The reachability-bound problem. In *PLDI*, 2010.

[Gupta *et al.*, 2011] A. Gupta, C. Popeea, and A. Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs. In *POPL*, pages 331–344, 2011.

[Gustafsson *et al.*, 2005] J. Gustafsson, A. Ermedahl, and B. Lisper. Towards a flow analysis for embedded system C programs. In *WORDS*, 2005.

[Gustafsson *et al.*, 2006] J. Gustafsson, A. Ermedahl, and B. Lisper. Algorithms for infeasible path calculation. In *WCET*, 2006.

[Healy and Whalley, 2002] C. A. Healy and D. B. Whalley. Automatic detection and exploitation of branch constraints for timing analysis. *IEEE Trans. Softw. Eng.*, 28(8):763–781, 2002.

[Henzinger *et al.*, 2002] T. A. Henzinger, T. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.

[Henzinger *et al.*, 2004] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, 2004.

[Hoare, 1969] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 1969.

[Hoffmann *et al.*, 2011] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. In *POPL*, 2011.

[Huynh *et al.*, 2011] B. K. Huynh, L. Ju, and A. Roychoudhury. Scope aware data cache analysis for WCET estimation. In *RTAS*, 2011.

[Ip and Dill, 1996] C. N. Ip and D. L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2):41–75, 1996.

[Jacobs and Piessens, 2008] B. Jacobs and F. Piessens. The Verifast Program Verifier, 2008.

[Jaffar *et al.*, 1992] J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The CLP($\mathcal{R}$) language and system. *ACM TOPLAS*, 14(3):339–395, 1992.

[Jaffar *et al.*, 1993] J. Jaffar, M. Maher, P. Stuckey, and R. Yap. Projecting CLP($\mathcal{R}$) constraints. *New Generation Computing*, 1993.

[Jaffar *et al.*, 2008] J. Jaffar, A. E. Santosa, and R. Voicu. Efficient memoization for dynamic programming with ad-hoc constraints. In *AAAI*, 2008.

[Jaffar *et al.*, 2009] J. Jaffar, A. E. Santosa, and R. Voicu. An interpolation method for clp traversal. In *CP*, 2009.

[Jaffar *et al.*, 2011] J. Jaffar, J.A. Navas, and A. Santosa. Unbounded Symbolic Execution for Program Verification. In *RV*, 2011.

[Joksch, 1966] H. C. Joksch. The shortest route problem with constraints. *Journal of Mathematical Analysis and Applications*, 14(2):191–197, 1966.

[Kahlon *et al.*, 2009] V. Kahlon, C. Wang, and A. Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *CAV*, 2009.

[King, 1976] J. C. King. Symbolic Execution and Program Testing. *Com. ACM*, 1976.

[Knuth, 1997] D. E. Knuth. *The art of computer programming.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[Li and Malik, 1995] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *DAC*, 1995.

[Lundqvist and Stenström, 1999] T. Lundqvist and P. Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *RTS*, 1999.

[Lv *et al.*, 2008] M. Lv, Z. Gu, N. Guan, Q. Deng, and G. Yu. Performance comparison of techniques on static path analysis of WCET. In *EUC*, 2008.

[Mälardalen, 2006] Mälardalen WCET research group benchmarks. URL `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`, 2006.

[Mazurkiewicz, 1986] A. W. Mazurkiewicz. Trace theory. In *Advances in Petri Nets*, 1986.

[McMillan, 2003] K. L. McMillan. Interpolation and SAT-based model checking. In *CAV*, 2003.

[McMillan, 2006] K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, 2006.

[McMillan, 2010] K. L. McMillan. Lazy annotation for program testing and verification. In *CAV*, 2010.

[NIST, 2002] Software errors cost u.s. economy $59.5 billion annually: NIST asseses technical needs of industry to improve software testing. URL http://www.cse.buffalo.edu/~mikeb/Billions.pdf, 2002.

[Park, 1993] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Syst.*, 5(1):31–62, 1993.

[Podelski and Rybalchenko, 2005] A. Podelski and A. Rybalchenko. Transition predicate abstraction and fair termination. In *POPL*, 2005.

[Prantl *et al.*, 2008] A. Prantl, J. Knoop, M. Schordan, and M. Triska. Constraint solving for high-level WCET analysis. *WLPE*, 2008.

[Puschner and Burns, 2000] P. Puschner and A. Burns. A review of worst-case execution-time analysis. *Journal of Real-Time Systems*, 2000.

[Reps *et al.*, 1995] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, 1995.

[Rybalchenko and Sofronie-Stokkermans, 2007] A. Rybalchenko and V. Sofronie-Stokkermans. Constraint solving for interpolation. In *VMCAI*, 2007.

[Saswat, 2012] A. Saswat. *Techniques to facilitate symbolic execution of real-world programs.* Ph. D. Thesis, Georgia Institute of Technology, 2012.

[Silva and Sakallah, 1996] J. P. M. Silva and K. A. Sakallah. Grasp-a new search algorithm for satisfiability. In *ICCAD*, 1996.

[Sinha and Wang, 2010] N. Sinha and C. Wang. Staged concurrent program analysis. In *FSE*, 2010.

[Sinha and Wang, 2011] N. Sinha and C. Wang. On interference abstractions. In *POPL*, 2011.

[Sistla and Godefroid, 2004] A. P. Sistla and P. Godefroid. Symmetry and reduced symmetry in model checking. *ACM TOPLAS*, 26(4):702–734, July 2004.

[SPIN,] SPIN model checker. URL http://spinroot.com.

[Stappert *et al.*, 2001] F. Stappert, A. Ermedahl, and J. Engblom. Efficient longest executable path search for programs with complex flows and pipeline effects. In *CASES '01*, pages 132–140. ACM, 2001.

[Suhendra *et al.*, 2006] V. Suhendra, T. Mitra A. Roychoudhury, and T. Chen. Efficient detection and exploitation of infeasible paths for software timing analysis. In *DAC '06: Proceedings of the 43rd annual Design Automation Conference*, pages 358–363. ACM, 2006.

[Thakur and Govindarajan, 2008a] A. Thakur and R. Govindarajan. Comprehensive path-sensitive data-flow analysis. In *CGO*, 2008.

[Thakur and Govindarajan, 2008b] A. Thakur and R. Govindarajan. Comprehensive path-sensitive data-flow analysis. In *CGO '08*, pages 55–63, 2008.

[Theiling *et al.*, 2000] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *RTS*, 2000.

[Theiling, 2002] H. Theiling. *CFGs For Real-Time Systems Analysis*. Ph. D. Thesis, 2002.

[Valmari, 1991] A. Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets*, pages 491–515, 1991.

[Verge, 1994] H. Le Verge. A note on chernikova's algorithm. Technical report, 1994.

[Vivancos *et al.*, 2001] E. Vivancos, C. Healy, F. Mueller, and D. Whalley. Parametric timing analysis. In *LCTES '01*, pages 88–93. ACM, 2001.

[Wahl and D'Silva, 2010] T. Wahl and V. D'Silva. A lazy approach to symmetry reduction. *Form. Asp. Comput.*, 2010.

[Wahl, 2007] T. Wahl. Adaptive symmetry reduction. In *CAV*, 2007.

[Wang *et al.*, 2008a] C. Wang, Y. Yang, A. Gupta, and G. Gopalakrishnan. Dynamic model checking with property driven pruning to detect race conditions. In *ATVA*, 2008.

[Wang *et al.*, 2008b] C. Wang, Z. Yang, V. Kahlon, and A. Gupta. Peephole partial order reduction. In *TACAS*, 2008.

[Wang *et al.*, 2009] C. Wang, S. Chaudhuri, A. Gupta, and Y. Yang. Symbolic pruning of concurrent program executions. In *ESEC/FSE*, 2009.

[Wilhelm *et al.*, 2008] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *Trans. on Embedded Computing Sys.*, 2008.

[Wilhelm, 2004] R. Wilhelm. Why AI + ILP is good for WCET, but MC is not, nor ILP alone. In *VMCAI*, 2004.

[Yang *et al.*, 2008] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby. Efficient stateful dynamic partial order reduction. In *SPIN*, 2008.