The Dissertation Committee for Jyotirmoy Vinay Deshmukh
certifies that this is the approved version of the following dissertation:

Verification of Sequential and Concurrent Libraries

Committee:

E. Allen Emerson, Supervisor

Adnan Aziz, Supervisor

Craig M. Chase

Vijay K. Garg

Sarfraz Khurshid

Aloysius K. Mok

Verification of Sequential and Concurrent Libraries

by

Jyotirmoy Vinay Deshmukh, B.E.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2010

I dedicate this dissertation to my family. To Aai, who made me who I am. To Urmila, who helped me re-discover myself. To Baba, who has taught me humility, and Aditi who has always been supportive. Finally, to my darling son Shashvat, who after taking his first breath, stared at me with his big, lovely eyes full of the accusation, "You are still a student? Graduate before I can walk, and I will forgive you."

Acknowledgments

As a fresh bachelor of electronics engineering, I began my explorations into the world of formal methods in computer science with a sense of wide-eyed wonderment, with little knowledge on how to be a successful academic or a researcher. I can hardly express how much of an impact my teachers and colleagues have had on me. The foremost among them is my primary research advisor, Prof. E. Allen Emerson. Among the various things that I have learnt from Allen, the most important has been his emphasis on rigor and discipline in the approach to any problem. I must mention that for his invention of model checking, Allen was recognized by the ACM A. M. Turing Award in 2007, the most prestigious award in Computer Science. What makes Allen such a delight to work with is that under the trappings of such an imminent computer scientist, he is a truly nice and humble man. Over the past few years, we have had several light-hearted chats about everything from literature in science fiction to political leanings. His honesty, respectfulness and, above all, friendliness always allow me to be comfortable expressing my opinions. I will miss working with Allen, who has been a teacher, a mentor, and a friend.

I am grateful to Prof. Aziz, my co-advisor for his extremely detailed reviews for my dissertation. In the course of my degree, he has always given

sound advice on how to present my research, and on the right practical goals to pursue. The seed for the first part of my dissertation emerged in a class taught by Prof. Sarfraz Khurshid. I am thankful to him for his suggestions on the kind of ideas to pursue for data structure verification, and for his keen interest in the progress of my degree. I am indebted to Prof. Garg for providing me with an opportunity to work as his research assistant, exposing me to important problems in concurrent and distributed systems. On a lighter note, my research in automatic techniques in concurrent programming spiked sharply when as a teaching assistant, I helped him grade student assignments in concurrent programming. I have also had the pleasure of working as a teaching assistant for Prof. Craig Chase. Through several interesting discussions, he has sharpened my sensitivity towards the engineering aspects of programming, including their ramifications in language design and features. I am indebted to Prof. Al Mok for his service as my committee member.

I am highly grateful to Sriram Sankaranarayanan, who has been an exemplary collaborator. Technical discussions with him are a continuous free-flow of ideas, and his excitement to work on a new idea is always infectious. Above all, he has selflessly offered invaluable advice that has helped me in steering my academic and professional pursuits in the right direction. I thank my colleagues Vineet Kahlon and Thomas Wahl for several interesting discussions. Vineet, who is now a researcher at NEC Labs, gave me plenty of guidance in the course of my internship with his group. I enjoyed brainstorming ideas with Thomas, whose methodical approach to even informal discussions

was really instructive.

I am indebted to the support staff at CERC: Linda Frost, Andrew Kieschnick, Melissa Campos, Debi Prather, and Melanie Gulick in ECE who helped me navigate the intricacies of various administrative matters. I thank Sankar Gurumurthy, Ramyanshu Datta and Sriram Sambamurthy for all their support and more importantly for their company to various coffee shops during my stay at CERC.

I owe a lot to Narendra Kamat and Roopsha Samanta, my extended family in Austin. Naren and Roopsha have always been there for me and my family, to lend a helping hand in times of trouble. I have known Naren since 1996, and having him be in the same city has been a source of immense comfort. Roopsha, who has also been my colleague, is an excellent person to bounce my ideas off, and has always given me constructive criticism that has helped me refine my ideas.

I thank my friends Abhay Pradhan, Deepak Agarwal, Milind Nagda, Dhiraj Acharya, Poorna Samanta and Nishit Gujral who as friends and roommates tolerated my extremely odd hours and various idiosyncrasies. I thank my friend Devdutt Nayak, who was probably the first person to teach me the importance of clarity in thought.

Without my family, none of what I have accomplished would have been possible. My mother, Nandini Deshmukh, through her tireless efforts ensured that I always worked to my potential, and had the best resources available to

enable me to succeed. Thank you Aai. My father, Vinay Deshmukh, taught me the importance of humility and ethics in my career and life. My sister Aditi, through dedication to her field of veterinarian sciences has served as an inspiration to me. I thank my uncle, Mahesh Nerurkar, who first interested me in mathematics; and my family - Rajan and Sandhya Deshmukh, and Anil and Shama Kulkarni, who have always lent me tremendous support through my stay in USA.

The companionship of my wife, Urmila Patil, has made the trials and tribulations of the life as a graduate student, an experience to cherish throughout my life. Her exemplary perseverance and focus have left a deep impression on me; in many ways her zeal towards her own research made me much more energetic about my approach towards my work. The small and large things that she has assisted me in are too many to enumerate. I thank my son, Shashvat, for being such an utter delight – an endless source of amusement, entertainment, and joy, in the daily routine.

Verification of Sequential and Concurrent Libraries

Publication No. _____

Jyotirmoy Vinay Deshmukh, Ph.D.
The University of Texas at Austin, 2010

Supervisors: E. Allen Emerson
Adnan Aziz

The goal of this dissertation is to present new and improved techniques for fully automatic verification of sequential and concurrent software libraries. In most cases, automatic software verification is plagued by undecidability, while in many others it suffers from prohibitively high computational complexity. Model checking – a highly successful technique used for verifying finite state hardware circuits against logical specifications – has been less widely adapted for software, as software verification tends to involve reasoning about potentially infinite state-spaces. Two of the biggest culprits responsible for making software model checking hard are heap-allocated data structures and concurrency.

In the first part of this dissertation, we study the problem of verifying *shape* properties of sequential data structure libraries. Such libraries are implemented as collections of *methods* that manipulate the underlying data structure. Examples of such methods include: methods to insert, delete, and

update data values of nodes in linked lists, binary trees, and directed acyclic graphs; methods to reverse linked lists; and methods to rotate balanced trees. Well-written methods are accompanied by documentation that specifies the observational behavior of these methods in terms of pre/post-conditions. A pre-condition φ for a method $\mathcal{M}$ characterizes the state of a data structure before the method acts on it, and the post-condition ψ characterizes the state of the data structure after the method has terminated. In a certain sense, we can view the method as a function that operates on an input data structure, producing an output data structure.

Examples of such pre/post-conditions include shape properties such as acyclicity, sorted-ness, tree-ness, reachability of particular data values, and reachability of pointer values, and data structure-specific properties such as: "no red node has a red child", and "there is no node with data value '`a`' in the data structure". Moreover, methods are often expected not to violate certain safety properties such as the absence of dangling pointers, absence of null pointer dereferences, and absence of memory leaks. We often assume such specifications as implicit, and say that a method is incorrect if it violates such specifications. We model data structures as directed graphs, and use the two terms interchangeably. Verifying correctness of methods operating on graphs is an instance of the *parameterized verification* problem: for *every* input graph that satisfies φ, we wish to ensure that the corresponding output graph satisfies ψ. Control structures such as loops and recursion allow an arbitrary method to simulate a Turing Machine. Hence, the parameterized verification problem

for arbitrary methods is undecidable.

One of the main contributions of this dissertation is in identifying mathematical conditions on a programming language fragment for which parameterized verification is not only decidable, but also efficient from a complexity perspective. The decidable fragment we consider can be broadly sub-divided into two categories: the class of iterative methods, or methods which use loops as a control flow construct to traverse a data structure, and the class of recursive methods, or methods that use recursion to traverse the data structure.

We show that for an iterative method operating on a directed graph, if we are guaranteed that if the number of destructive updates that a method performs is *bounded* (by a constant, *i.e.*, $O(1)$), and is guaranteed to terminate, then the correctness of the method can be checked in time *polynomial* in the size of the method and its specifications. Further, we provide a well-defined *syntactic* fragment for recursive methods operating on tree-like data structures, which assures that any method in this fragment can be verified in time *polynomial* in the size of the method and its specifications. Our approach draws on the theory of tree automata, and we show that parameterized correctness can be reduced to emptiness of finite-state, nondeterministic tree automata that operate on infinite trees. We then leverage efficient algorithms for checking the emptiness of such tree automata to obtain a tractable verification framework. Our prototype tool demonstrates the low theoretical complexity of our technique by efficiently verifying common methods that operate on data structures.

In the second part of the dissertation, we tackle another obstacle for tractable software verification: concurrency. In particular, we explore application of a static analysis technique based on interprocedural dataflow analysis to predict and document deadlocks in concurrent libraries, and analyze deadlocks in clients that use such libraries. The kind of deadlocks that we focus result from circular dependencies in the acquisition of shared resources (such as locks). Well-written applications that use several locks implicitly assume a certain partial order in which locks are acquired by threads. A cycle in the lock acquisition order is an indicator of a possible deadlock within the application.

Methods in object-oriented concurrent libraries often encapsulate internal synchronization details. As a result of information hiding, clients calling the library methods may cause thread safety violations by invoking methods in a manner that violates the partial ordering between lock acquisitions that is implicit within the library. Given a concurrent library, we present a technique for inferring *interface contracts* that specify permissible concurrent method calls and patterns of aliasing among method arguments that guarantee deadlock-free execution for the methods in the library. The contracts also help client developers by documenting required assumptions about the library methods. Alternatively, the contracts can be statically enforced in the client code to detect potential deadlocks in the client. Our technique combines static analysis with a symbolic encoding for tracking lock dependencies, allowing us to synthesize contracts using a satisfiability modulo theories (SMT) solver. Additionally, we investigate extensions of our technique to reason about dead-

locks in libraries that employ signaling primitives such as *wait-notify* for cooperative synchronization. We demonstrate its scalability and efficiency with a prototype tool that analyzed over a million lines of code for some widely-used open-source Java libraries in less than 50 minutes. Furthermore, the contracts inferred by our approach have been able to pinpoint real bugs, *i.e.*, deadlocks that have been reported by users of these libraries.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Part I

Introduction

Chapter 1

Introduction

1.1 Motivation

Software systems now control several important aspects of human life. The benefits of automation come at a price; we have to endure the possibility of catastrophic system failures resulting from programming errors. In the recent past, crippling bugs have surfaced in the software that governs the operation of life-support systems, medical devices, space missions, safety features in cars, and microprocessor chips. Even in mundane software systems, errors severely hamper the productivity of the users of such software. A study prepared in 2002 for the National Institute of Standards and Technology[1] helps put things into perspective. It concluded that errors in software industry were costing the American economy $59 billion annually, of which $22.2 billion could be eliminated by earlier and more effective identification and removal of software defects.

Human fallibility in writing programs is likely to persist; thus, the onus lies with computer scientists to develop frameworks that are (a) capable

[1] http://www.nist.gov/public_affairs/taglance/taglance_summer02/summer2002.htm#bugs

of minimizing the likelihood of such errors, (b) verifying the correctness of existing programs, and (c) aid the debugging process by better identification of the root cause of bugs. Formal methods for synthesis, verification and debugging have long held the promise to help achieve these goals [74, 122, 46].

Automatic and semi-automatic techniques for ensuring program correctness or reliability can be sub-divided into two broad categories: dynamic techniques and static techniques.

Dynamic techniques operate by executing a program, and observing the results of the program execution. Testing and profiling of software are examples of dynamic analyses. Software testing runs the program against a suite of test-cases that encapsulate intended program behavior, and program correctness is predicated upon the program satisfying each of the test-cases within this suite. Testing usually occurs at different levels; unit testing for checking the functional correctness of individual modules; integration testing for checking the module interfaces versus the system design; and system testing for checking a completely integrated system against its requirements. The advantages of dynamic analyses are accuracy and speed (they are as fast as program execution), while the shortcomings are lack of coverage and dependence on human input for test-cases or code instrumentation for profiling. Though coverage tools help measure the amount of code coverage using various metrics, it is rare for a dynamic technique to offer comprehensive code coverage.

Static techniques on the other hand inspect the program code without

actually executing it, and reason about possible behaviors that may manifest at run-time. Static program analysis [91, 114], model checking [52, 30, 123], symbolic simulation [23], symbolic trajectory evaluation [151] are examples of static techniques.

In some cases, the efficacy of static techniques is often predicated upon the availability of specifications, which may require additional effort in development and formalization. Static techniques also make use of conservative abstractions, which lead to false positives (*i.e.*, errors introduced due to the abstraction process). Applications such as testing user interfaces (UI)s are also not easily amenable to analysis by static techniques. In spite of these limitations, static techniques hold the promise of exhaustive analysis of the program, by inspecting all possible program behaviors, and hence have increasingly become a popular alternative to dynamic techniques.

The choice of the right static technique is usually a tradeoff between precision and speed. Precise static techniques such as model checking [52, 30, 123] rely on statically constructing the state-space of the program. Model checking can be thought of as an intelligent search algorithm that scans the reachable configurations of the underlying system for certain temporal patterns that encode desirable or undesirable sequences of configurations. As state-spaces are often enormous, model checking faces the *state explosion* problem. In spite of this, due to advances such as symbolic model checking [107], bounded model checking [28], abstraction-refinement [29], symmetry reductions [27, 60], and partial order reduction [119, 67], model checking has fast become an efficient

approach to identify critical bugs early in the hardware development cycle. As testing-based approaches are inexhaustive, bugs that remain undiscovered till the post-silicon phase incur significant expense in both in terms of time and money, and hence model checking has endeared itself to the hardware verification community.

It can thus seem mystifying that model checking has not had as much of an impact on software verification. The key reason for this is the computational complexity of software verification in comparison to that of hardware. Even large hardware systems are essentially finite state, and with techniques to ameliorate the state explosion problem, model checking can now handle reasonably sized hardware designs. Software model checking, on the other hand, often runs into the brick wall of undecidability or extremely high complexity, often rendering it infeasible for practical software systems. Heap-allocated dynamic data structures, recursion, and program variables with large domains are commonplace in even the most ordinary software. Each of these in itself, often leads to either undecidability in reasoning, or causes prohibitively high number of program configurations that need to be inspected. Even successful software model checking tools such as SLAM [8], `Java PathFinder` [81] and BLAST [13] often use conservative approximations for heap-allocated structures, which may lead to a large number of false errors.

The faster static techniques, such as static program analysis, are inaccurate as they are prone to a slew of false positives, *i.e.*, errors arising from conservative approximation of program behavior that are infeasible during ac-

tual program execution. The challenge in these techniques is to increase the accuracy by novel approaches such as symbolic reasoning, without compromising on their space and time complexity.

Concurrency adds yet another dimension of complexity to static techniques in software verification. The increasing use of multi-threaded code is a major source of *heisenbugs*. Such bugs (named after the uncertainty principle) are particularly tricky to detect, as they only appear in certain interleavings of the multi-threaded code. The key reason for this is that individual threads or processes within concurrent programs are often scheduled by the operating system, giving the software developer little or no control over allowable concurrent behaviors – some of which may contain lurking heisenbugs. Therefore, a reasonable assumption in analyzing most multi-threaded or multi-process programs is thus that scheduling is nondeterministic, and it is possible to get all possible concurrent interleavings as possible program behaviors. Unfortunately, there is an exponential number of such interleavings to be considered, leading to an astronomical number of possible program configurations to be inspected. Coupled with dynamic data structures and recursion, reasoning about even the simplest concurrent programs becomes undecidable [124].

The overarching goal of this dissertation is development of strategies to make static techniques in software verification tractable. We accomplish this goal in two ways:

1. We formulate a framework for efficient verification of methods in se-

quential data structure libraries. This framework can be viewed as an extension of conventional model checking to automatically reason about methods that manipulate parameterized data structures.

2. We explore applications of static program analyses for ensuring thread safety in concurrent libraries. We introduce a novel symbolic encoding scheme for deadlock prediction in concurrent libraries, which is both more accurate than existing techniques, and highly scalable – being able to analyze over a million lines of `Java` code in less than 50 minutes, on an off-the-shelf (dual core) uniprocessor system.

We now introduce each of these sub-problems, with emphasis on why the problems are important, what makes them difficult to solve, and a brief overview of how we tackle them.

1.2 Verification of Data Structure Libraries

Problem Overview. Data structures are the basic building blocks for essentially all software systems. Most data structures are usually embedded in a data structure library and are associated with specialized *methods* that perform various operations on the data structures. Examples of such data structures include linked lists, queues, binary search trees, balanced trees, hash-tables, and general directed graphs. Examples of methods include methods to insert and delete nodes, methods to rotate parts of trees; and methods to reverse lists. It is often convenient to model data structures as vertex and edge-

labeled directed graphs, and methods as routines to transform these graphs. In this dissertation, we often use the terms data structures and directed graphs interchangeably.

The correctness of methods is often specified informally with Application Programming Interface (API) rules. These rules can be interpreted or re-written as pre/post-conditions, similar to Hoare logic [83]. If $\mathcal{M}$ is a method of interest, we denote by φ the pre-condition, which is a structural property of the input data structure (G_i) for a method. We often term the data structure resulting from the action of $\mathcal{M}$ on G_i as the output structure G_o, and denote it as $\mathcal{M}(G_i)$. We denote by ψ the post-condition or a structural property specifying the data structure resulting from the action of $\mathcal{M}$ on the input structure. The correctness problem for such methods can be stated as: "If each input data structure G_i for $\mathcal{M}$ satisfies φ, does the data structure G_o resulting from the action of $\mathcal{M}$ on G_i (denoted $\mathcal{M}(G_i)$) satisfy ψ?"

For instance, we would like to ensure that a method that inserts a node in a singly-linked list does not violate the invariant of acyclicity, or a method to delete a node from a balanced tree ensures that the resultant tree is also balanced, or a method that sorts a linked list produces a sorted list with elements that are a permutation of the original list. Frequently, the only correctness criterion required is that a method does not violate a structural invariant of the data structure, *i.e.*, the pre-condition and the post-condition are identical. Common examples of pre/post-conditions or structural invariants include:

1. *Shape* properties such as: acyclicity, existence of sharing (two nodes point to a common node), tree-ness (each non-root node has a unique parent), list-ness (given graph is a list), and balanced-ness (given tree is balanced).

2. Connectivity properties such as: reachability of a target node from a source node (where the nodes are specified by pointers), reachability of a given data value from a given node, unreachability of garbage (memory that has been marked as free), strongly connectedness (every node is reachable from itself).

3. Data-dependent properties such as sorted-ness, "no red node has a red child", "every node labeled b is preceded by a finite number of nodes all labeled a", and so on.

Methods that modify data structures usually perform a number of memory-related operations. Hence, an often unstated requirement is that the method should not contain violations of safety properties such as null pointer dereferences, presence of dangling pointers, and memory leaks. In most cases, in addition to the structural invariants specified above, we assume that invariance of safety properties is implicit to the verification problem.

In practice, programmers usually certify correctness of methods by testing them for candidate input data structures of varying sizes. However, data structures can get arbitrarily large during program execution, and it is computationally infeasible for a testing-based strategy to exhaustively verify cor-

rectness. In a certain sense, verification of data structure-altering methods is an instance of the *parameterized verification* problem [57, 58]. In parameterized verification, the system of interest is characterized by certain parameters, and correctness mandates that the system satisfies its specifications for all values that the parameters can take. Similarly, correctness of methods on data structures requires that the methods display desired behavior for all sizes of the input data structure, and not just for some sizes as in testing-based approaches. However, the biggest obstacle to the automatic verification of such methods is that parameterized verification is undecidabla [57, 58].

Solution Overview. Though verification of arbitrary methods is undecidable, we show that with a few restrictions, broad classes of practical methods on data structures can be algorithmically verified. Specifically, in this dissertation, we make the observation that under certain constraints, methods acting on data structures can be exactly mimicked by finite state machines. This critical insight allows us to employ an automata-theoretic framework to verify the parameterized correctness of such methods.

We show that for a method $\mathcal{M}$ that performs a bounded (by a constant k) number of destructive updates to the underlying data structure, we can construct an *exact* abstraction known as the *method automaton* $\mathcal{A}_{\mathcal{M}}$. The automaton $\mathcal{A}_{\mathcal{M}}$ is defined to operate on a *composite graph* G_c which is a superposition of a finite number of graphs $G_0, \ldots, G_k$, where G_0 is the input graph, the k pairs (G_0, G_1), ..., (G_{c-1}, G_l) encode the c destructive updates, and G_k

is the output graph. In other words, $G_1, \ldots, G_{k-1}$ denote intermediate values encountered during the action of $\mathcal{M}$ before obtaining the final output graph G_k.

We also show how a significant subset of the properties outlined before can be modeled using finite state nondeterministic tree automata on finite and infinite trees. We assume that pre/post-conditions are specified as a pre-condition automaton $\mathcal{A}_\varphi$ and a (negated) post-condition automaton $\mathcal{A}_{\neg\psi}$. The final step is to compute $\mathcal{A}_p$, the composition of $\mathcal{A}_\mathcal{M}$, $\mathcal{A}_\varphi$ and $\mathcal{A}_{\neg\psi}$, which is nonempty iff there exists an input graph satisfying φ for which the action of $\mathcal{M}$ produces an output graph that does not satisfy ψ. In effect, we reduce checking correctness of $\mathcal{M}$ to checking the language emptiness for $\mathcal{A}_p$. Thus, the final computational complexity of our approach is dominated by the complexity of nonemptiness of the product automaton, which is *polynomial* in the size of $\mathcal{A}_p$. This in turn is proportional to the sizes of the pre/post-condition automata, and the size of the method. Thus, the overall computational complexity is *polynomial* in the size of the method and its specifications.

In a certain sense, our basic methodology can be viewed as an extension of conventional model checking. The core characteristic of model checking, *i.e.*, fully automatic and efficient verification, given a set of specifications, is preserved by our approach.

Organization. In Chapter 2, we formally define data structures, the use of directed graphs to model data structures, and methods over parameterized

data structures. We present a finitary encoding (known as the window-based encoding) that allows data structures containing arbitrary pointer values to be encoded as sequences of finite neighborhoods within the data structure known as *windows*. Finally, we give the necessary background on tree automata to conclude the preliminaries.

Within the broad automata-theoretic framework for verifying methods that manipulate data structures, we consider two distinct classes of methods. In Chapter 3, we discuss iterative methods operating on general directed graphs. We establish mathematical conditions under which methods can be mimicked by finite state automata. We then outline Bud-Pl: a programming language with syntax similar to high-level programming languages such as `C` for expressing methods, and provide an algorithm to compile a method in Bud-Pl into a method automaton $\mathcal{A}_{\mathcal{M}}$.

In Chapter 4, we extend Bud-Pl to allow expression of recursive methods on tree-like data structures. In our approach, a method $\mathcal{M}$ is mimicked by a method automaton $\mathcal{A}_{\mathcal{M}}$. $\mathcal{A}_{\mathcal{M}}$ is defined to operate over some composite graph G_c (as defined before) that encodes the possible actions of some method. If the actions encoded in G_c are consistent with the actions of $\mathcal{M}$, then $\mathcal{A}_{\mathcal{M}}$ accepts G_c. Typically, recursive methods employ an unbounded stack to store information about the calling context before executing the recursive call. However, we show that a suitably defined composite graph contains all the relevant information about the calling context (*i.e.*, the state of the data structure) before a recursive call is executed. Thus, by constructing a suitable

composite graph, we can avoid explicit reasoning about the unbounded stack.

This syntactic class of recursive methods is an improvement over the class for iterative methods: methods in this class are guaranteed to respect the stipulations that allow them to be mimicked by finite state automata. Thus, methods in this class can be efficiently verified without further manual input. For expository reasons, we divide the decidable syntactic class of recursive methods into tail-recursive methods and the more general case, and give algorithms to translate each class of methods into method automata. Note that tail-recursive methods mimic iterative methods, and thus this extension also provides us with a syntactic class of iterative methods on tree-like data structures that can be automatically verified.

Our approach has favorable theoretical complexity. The practical utility of our approach is further demonstrated by our prototype tool PRAVDA, which is able to verify a large subset of both iterative and recursive methods on data structures, for many interesting shape properties, within reasonable time and memory constraints. We present the experimental results in Chapter 5.

1.3 Analysis of Concurrent Libraries

Problem Overview. There are two kinds of paradigms for parallel computation: shared memory systems (that we use synonymously with the term concurrent systems), and distributed systems. Distributed systems are characterized by the lack of a shared memory, and as a result it is impossible for any one process to know the global state of the system. Such systems thus use the

notion of a distributed memory, and use message passing for communication between different processors [68]. In this dissertation, we largely focus on the former paradigm, *i.e.*, concurrent systems or systems with shared memory.

Analysis of shared memory systems has gained considerable attention recently, especially since multi-core processors have become the *de facto* standard for hardware platforms. The day when systems with thousands of cores become ubiquitous is imminent. In spite of decades of research in the development of concurrent software, there is a gap between the programming practices for concurrent software and the theoretical research in concurrency synthesis and verification. This is evident by the number of errors arising in applications using poorly designed concurrent libraries. Such errors are commonly referred to as *thread safety violations*. We can broadly classify them into violations of safety properties such as *data races*, *atomicity violations* and violations of liveness properties such as *starvation* and *deadlocks*.

As safety violations such as data races are abundant and difficult to debug, they have garnered considerably more attention. A *knee-jerk* response to avoiding race conditions is evident in the prolific use of locking constructs in concurrent programs. Languages such as `Java` have promoted this by providing a convenient `synchronized` construct to specify mutual exclusion with monitors. Locking is sometimes naively used as a "safe" practice, rather than as a requirement. Overzealous locking not only causes unnecessary overhead, but can also lead to unforeseen deadlocks. Deadlocks can severely impair real-time applications such as web-servers, database systems, mail-servers, device

drivers, and mission-critical systems with embedded devices, and typically culminate in loss of data, unresponsiveness, or other liveness violations.

Deadlock detection is a well-studied problem, and approaches based on model checking, static analysis, dynamic (run-time) analysis have been applied for detecting deadlocks. Though model checking based tools can be quite useful for detecting deadlocks, for large software systems their scalability is often called into question. Hence, the programming languages community has gravitated towards the use of both static program analysis and dynamic analysis based techniques [79, 2, 32, 145, 5, 110, 148] for deadlock detection. Typically, such techniques construct *lock-acquisition order graphs* that track dependencies between locks for each thread. Lock-order graphs for concurrent threads are then merged, and a cycle in the resulting graph indicates a possibility of a deadlock. Such techniques typically assume a *closed system*, and are thus useful for detecting *existing deadlocks* in a given application.

However, most software is modular and treating individual components as closed systems could lead to potential deadlocks being undetected. In particular, consider the now prevalent *concurrent libraries*, *i.e.*, collections of modules that support concurrent access by multiple client threads. Modular design principles mandate that the onus of ensuring thread safety lies with the developer of such a library. This has an undesirable side-effect: several details of synchronization are obscured from the developer of client code that makes use of this library. Consequently, the client developer may unintentionally invoke library methods in ways that can cause deadlocks.

In the later part of this dissertation, we investigate a kind of static program analysis that we have called *deadlockability analysis*. The goal of deadlockability analysis is to use static analysis for the prediction, documentation, and analysis of deadlocks in concurrent libraries. A form of deadlockability analysis was first investigated in [148]. However, the authors used *types* of syntactic expressions corresponding to object monitors as conservative approximations for the alias information between these monitors. While the authors identify important potential deadlocks, their approach is susceptible to many false positives, which have to be then filtered using (possibly unsound) heuristics. Higher precision can be attained if precise information about aliasing between object monitors is available. However, the cost of performing a precise alias analysis and recording the high number of aliasing relationships that can exist between monitors can add up, affecting the scalability of the deadlockability analysis.

Solution Overview. We focus on deadlocks arising from circular dependencies in lock acquisition in common concurrent programming languages. We also identify usage patterns of *wait-notify* based synchronization that can lead to potential deadlocks.

In our approach, we reason about possible aliasing patterns between nodes in lock graphs explicitly rather than with type-based approximations. As suspected, this incurs a considerable cost, as several complex aliasing relationships have to be remembered and used by our tool. In fact, we show

that the problem of finding all aliasing relationships between nodes of two lock-order graphs, such the aliasing relationship causes the merged lock-order graph to have a cycle is NP-complete by a reduction from CNF-SAT.

However, we reduce the space of possible aliasing patterns between nodes by using a notion of subsumption between aliasing patterns. Essentially, we show that if an aliasing pattern α causes a deadlock, then every aliasing pattern that subsumes α also causes a deadlock. Conversely, we show that if an aliasing pattern α is *safe*, then every aliasing pattern subsumed by α is also safe.

We also introduce a symbolic encoding for lock-order graphs and aliasing relationships that encodes the cycle detection problem as a constraint satisfaction problem. We then feed the set of constraints to a *satisfiability modulo theories* (SMT) solver, which uses various engineering optimizations, such as incremental cycle detection and unsatisfiable cores, to efficiently enumerate all aliasing patterns that lead to deadlocks. By enabling symbolic reasoning along with subsumption, our analysis is highly scalable. The focus on aliasing patterns allows us to rule out infeasible aliases by means of a prior pointer analysis. Thus, our analysis combines the scalability of static analysis with the precision of an approach based on model checking or dynamic analysis.

Lastly, we synthesize logical expressions, termed *interface contracts* involving aliasing between the parameters of concurrent method invocations such that these expressions guarantee deadlock-free execution of the library methods. These contracts can then be used to statically detect deadlocks or dy-

namically enforce deadlock freedom in a particular client.

Organization. In Chapter 6, we introduce the general principles of static analysis and program analysis techniques. We formally introduce a programming language model that closely adheres to object-oriented imperative languages such as `Java` and `C++`. We include a discussion on the implementation of common synchronization primitives such as locks, monitors and condition variables within these languages. We conclude with an informal introduction to deadlockability analysis.

In Chapter 7, we discuss a static analysis algorithm to compute lock-graphs for methods in a concurrent library. We define deadlockability analysis, and finally present a symbolic encoding to represent lock-graphs and aliasing patterns as constraints in a suitable decidable theory. We then show how the symbolic encoding can be leveraged to efficiently enumerate all deadlock-causing aliasing patterns for methods in a library using a SMT solver. Finally, we discuss how the results of deadlockability analysis for a library can be used for statically detecting deadlocks in a client.

In Chapter 8, we extend deadlockability analysis to methods that use signaling-based synchronization with the help of *wait-notify* statements. We extend the nested monitors rule used for deadlock analysis in lock-based synchronization, and term it the generalized nested monitors rule. We show how the static algorithm for lock-graph computation can be extended to accommodate the generalized rule. The resultant graph obtained from such a static

analysis is termed an extended lock-graph. Finally, we show how the extended lock-graph can be symbolically encoded into constraints, thereby enabling symbolic reasoning for enumerating deadlock-causing aliasing patterns as in Chapter 7.

An approach based on static analysis can produce too many false positives to be of practical use. However, as we use precise aliasing information, our approach does not suffer from this shortcoming as much. We empirically validate our technique with the help of a tool on a broad class of `Java` libraries. We present the most notable results in Chapter 9.

Any serious research is incomplete without situating it in the past and the contemporary work in the area. Hence, after each chapter we include a detailed discussion on related work. Lastly, we summarize our contributions, and enumerate a list of open problems in Chapter 10.

Part II

Verification of Sequential Data Structures

Outline. Sequential software verification is hard due to (1) variables with large domains such as integers, strings, and floating-point precision variables, (2) heap-allocated linked data structures that can grow arbitrarily in size such as queues, linked lists, and trees, and (3) unrestricted recursion. Each of these by itself makes software verification undecidable. There are two possible approaches to mitigate this issue: (a) develop algorithmic solutions that can conservatively approximate software correctness, (b) identify decidable subclasses of software programs that are amenable to exact verification.

In our approach to verifying programs that manipulate data structures, we choose Option (b). The goal of our work in Part II is thus to formulate (sufficient) mathematical conditions for programs that ensure efficient and exact software verification. In Chapter 3 we show how such conditions can be used to verify iterative methods on general directed graphs. In Chapter 4, we show how we present a well-defined syntactic fragment of recursive methods that can be automatically verified. In Chapter 5, we present experimental evaluation of the techniques developed in Chapters 3 and 4.

Chapter 2

Preliminaries

In this chapter, we introduce the necessary formal notation, definitions, and terminology for sequential verification of data structure manipulating programs.

2.1 Data Structure Manipulating Programs

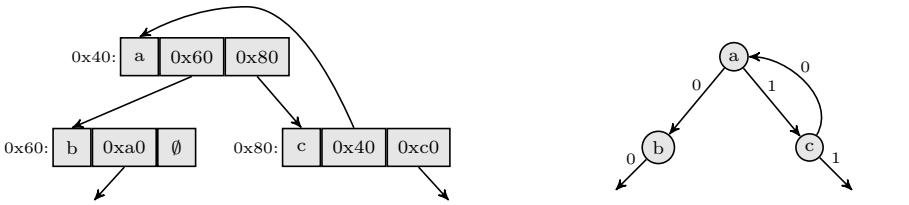
A data structure manipulating program can be described as a pair (S, V), where V is a set of program variables, and S is a set of program statements. We assume that program variables belong to only one of two types: a variable over some finite set $\mathcal{D}$ (referred to as a $\mathcal{D}$-variable), and a *pointer variable*. $\mathcal{D}$ is termed as the *data domain*: typical examples include the set of alphanumeric characters, the set of integers up to a fixed word length, the set of strings up to a fixed length over a finite alphabet, finite products of such sets, *etc.* To define a pointer variable, we first introduce the memory model.

Memory Model. A *heap* H is an ordered set (array) of memory cells, where each memory cell contains the value of some program variable. We say that

Ω is an address domain that contains memory addresses and `null`. Each memory cell has a unique address from the set $\Omega - \{\texttt{null}\}$. A *pointer variable* (or simply pointer) p is a variable that evaluates to some address in Ω. We seek to reason about heap-allocated data structures. In general, this consists of arrays (data structures with contiguous memory allocation) and linked data structures (non-contiguous). In this work, we focus on the latter.

The basic building block of a linked data structure is a *node*. A *node* n consists of contiguous memory locations containing values of the tuple of variables: $(d, l_1, \ldots, l_K)$. Note that Here d is a $\mathcal{D}$-variable, while each l_i is a pointer. Note that the number of links K is fixed and is part of the definition for a node. We use $\texttt{addr}(n)$ to denote the address of the first memory location within n. We use $n.d$ (respectively $n.l_i$) to denote the variable d (respectively pointer l_i) contained in n. The value $n.d$ is also called the *data value* of n, and $n.l_i$ is also referred to as a *link*, or the i^{th} *next pointer* of n. We say that a pointer p *points to* n if $p = \texttt{addr}(n)$. If $p = \texttt{addr}(n)$, then the expression $\texttt{deref}(p)$, also called *dereference* of p evaluates to n. If the value of p is `null`, the expression $\texttt{deref}(p)$ is undefined and generates an error known as a *null pointer dereference* (NPD). We assume that all pointers p in our programs are either `null`, or that $\texttt{deref}(p)$ evaluates to a valid node. Wherever convenient, we use the symbols $\emptyset$ and $\perp$ interchangeably with `null`.

Data Structure. A *data structure* is defined as an arbitrary set of nodes. In this work, we focus on *rooted* and *connected* data structures. As shown in



(a) Data Structure (b) Corresponding Directed Graph

Figure 2.1: Correspondence between a Data Structure and a Directed Graph

Figure 2.1, a rooted, connected data structure D has a one-to-one correspondence with a rooted, labeled, directed graph $G(V, E, \mathcal{L}_V, \mathcal{L}_E)$: V is the set of vertices, E is the set of edges, $\mathcal{L}_V : V \mapsto \mathcal{D}$ is a function that maps each vertex to some data value in $\mathcal{D}$, and $\mathcal{L}_E : E \mapsto \{1, \ldots, K\}$ is a function that maps each out-going edge of a given node to a unique positive integer. Each node n in D corresponds to a vertex v_n, such that $\mathcal{L}_V(v_n) = n.d$, and for every n', such that $n.l_i = addr(n')$, the edge $e : (v_n, v_{n'})$ belongs to E, and $\mathcal{L}_E(e) = i$. We term every v' such that $(v, v') \in E$, a *child* or successor of v, and term v a *parent* or predecessor of v'. We assume that there is a designated unique *root* vertex $r \in V$ (corresponding to the root node of the data structure) with the property that r has no predecessor.

Parameterized Data Structures and Shapes. A specific data structure is characterized by certain structural properties that can loosely be termed the *shape* of the data structure. For instance, an acyclic singly linked list is characterized by the properties that: (a) each node can have at most one successor, (b) each node has a unique parent (except the root node, which has

none), (c) starting from the root node of the list, it is possible to reach every node in the list by following the links to the successors, and finally reach a node that does not have a successor (*i.e.*, $n.l_0$ is `null`).

Several formalisms such as first-order logic [21], monadic second order logic [108], various decidable logical fragments [9, 152], three-valued logics [131], separation logic [125, 115] and tree automata [42, 20, 41] have been employed to describe shapes of data structures. In this work, we focus on the use of tree automata and fragments of temporal logic (such as *CTL* and μ-calculus) to describe such properties. Examples of specifications of such shape properties can be found in Chapter 3.

An important aspect of dynamic, heap-allocated data structures is that they typically do not have a fixed bound on their size, and can *grow* during the execution of the program. Thus, dynamic data structures can get arbitrarily large. As we wish to focus on verifying programs that alter such data structures, we are required to verify program correctness for all possible data structure sizes. It is thus useful to consider such a data structure as a *parameterized system*: different instances of the data structure can be uniformly described using a representation that is parameterized by attributes of the data structure. For example, an acyclic singly linked list is parameterized by its length, and a binary tree is parameterized by its height and/or by the number of nodes.

Let G denote the corresponding directed graph for a given data structure D. We denote by φ the logical formula (in any appropriate formalism)

describing a specific shape property over graphs. The parameterized *family* of graphs, denoted $\mathcal{G}_\varphi$ is described as the set $\mathcal{G}_\varphi = \{G \mid \varphi(G)\ is\ true\}$. We also use $G \models \varphi$ to denote that $\varphi(G)$ is *true*.

Methods over Parameterized Data Structures. In the object-oriented programming parlance, a method is a sub-routine associated with an abstract data type specified as a class or with an object that is an instance of such a class. In this work, we specialize the meaning of this term.

Def. 2.1.1 (Method)**.** A method $\mathcal{M}$ is a program that operates on an input data structure D_i, and (possibly) modifies it, resulting in the data structure D_o. We denote this as $D_o = \mathcal{M}(D_i)$[1].

An arbitrary method contains a set of $\mathcal{D}$-variables, and a set of pointer variables. We assume that each pointer variable points to some valid node of the input (or output) data structure. We use the terms *cursor* or *iterator* to denote such pointer variables. It is important to bear in mind that the methods that we seek to verify operate on parameterized data structures. Such data structures can get arbitrarily large; hence, methods typically employ control-flow structures such as loops or recursion for traversal over the structure. A *single pass* over the data structure is a traversal that visits each node exactly once. Methods perform traversal with the use of cursors, by advancing to

[1]It is possible to view a method as a partial function that maps an input data structure D_i to some output data structure D_o. Henceforth, we refer to the data structure resulting from the action of the method $\mathcal{M}$ as the output data structure for convenience.

nodes that are reachable from the current set of nodes (accessible through the current set of cursors) and modifying cursors to point to these "next" nodes. The general syntax and semantics of such methods closely mimics high-level programming languages such as `C++` or `Java`. We give the precise syntax and semantics in later chapters.

2.2 Problem Definition

Def. 2.2.1 (Parameterized Correctness)**.** Let φ be a shape property specifying valid input graphs. Let ψ be a shape property specifying valid output graphs. We wish to verify the total correctness assertion: $\langle\varphi\rangle\mathcal{M}\langle\psi\rangle$.

In other words, the above correctness assertion says that given families $\mathcal{G}_\varphi$ and $\mathcal{G}_\psi$, we wish to check if for all input graphs $G_i \in \mathcal{G}_\varphi$, $\mathcal{M}(G_i)$ is in $\mathcal{G}_\psi$. An alternate, equivalent, way to state the above problem is: if for any graph G_i s.t. $\varphi(G_i)$ is *true*, if there exists $G_o = \mathcal{M}(G_i)$, s.t. $\neg\psi(G_o)$ is *true*, then $\mathcal{M}$ fails to satisfy its correctness assertion. We refer to φ as the pre-condition, and ψ as the post-condition.

A special case of Def. 2.2.1 is invariance of shape properties. In this case, φ is the same as ψ, and we wish to verify the total correctness assertion: $\langle\varphi\rangle\mathcal{M}\langle\varphi\rangle$.

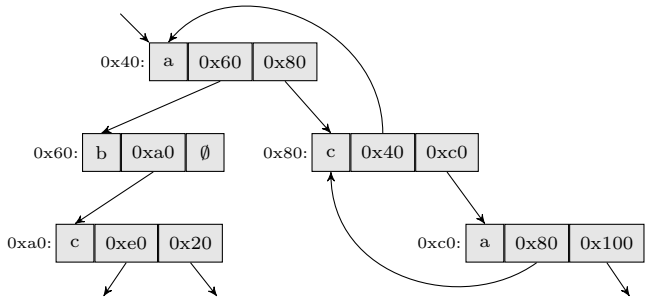
2.3 Window-based Encoding

As data structures can have arbitrary sizes, the addresses of the nodes in the data structure are unbounded. This is one of the key reasons why reasoning about data structures fails to scale when reasoning explicitly about the program state. Arbitrarily large addresses also lead to undecidability as we see in Chapter 3.

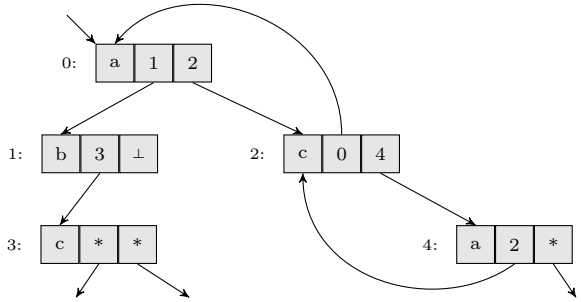
Hence, we propose a finite encoding for a data structure that abstracts a sequence of nodes in the data structure as a sequence of *local neighborhoods* within the data structure. Each such local neighborhood is an abstraction of the corresponding set of nodes in the data structure. We refer to such a local neighborhood as a *window*.

Formally, a window w is a finite encoding of a node n and a bounded number of nodes that succeed n, similar to the finite encodings developed in [42, 19]. Let $G(V, E)$ be a directed graph corresponding to some data structure D. Let Ω denote the set of memory addresses for the nodes in D. A window is obtained by mapping memory addresses within the neighborhood of a designated node to a small, finite subset of integers.

Def. 2.3.1 (Window)**.** Given a node u_0, let $nodes(u_0, z) = \{u_1, \ldots, u_\ell\}$ be the set of nodes within distance z from u_0, *i.e.*, $\forall i$, (u_0, u_i) is in one of E or E^2, ..., E^z, for some (small) finite positive integer z. A *window* of height z rooted at u_o (denoted $w_z(u_0)$) is the set $\{\hat{u_0}, \ldots, \hat{u_\ell}\}$, where: If $u_i = (d_i, addr(u_{i_1}), \ldots, addr(u_{i_K}))$, then, $\hat{u_i} = (d_i, la(u_{i_1}), \ldots, la(u_{i_K}))$. Here, la or



(a) Neighborhood in a Concrete Data Structure



(b) Window of Diameter 2

Figure 2.2: Window-based Abstraction

the *local address* function replaces the concrete memory address for a node u_i with i if $u_i \in nodes(u_0, z)$, and $*$ otherwise.

Intuitively, given a small neighborhood of nodes (of size ℓ) with a designated root node u_0, a window assigns an integer $0, \ldots, \ell - 1$ as an abstract address to each node in the neighborhood. It then proceeds to replace the concrete addresses in the links of the nodes within this neighborhood with the abstract addresses. If a link points to a node outside the neighborhood, the corresponding abstract address is marked as $*$.

We note that this construction can be extended to allow for predecessor nodes of u_0. The idea is to designate u_0 as the "center" of the window, with abstract addresses that can range over negative integers (indicating predecessors), zero (indicating u_0) and postive integers (indicating successors). Figure 2.2 gives an example of the window-based abstraction.

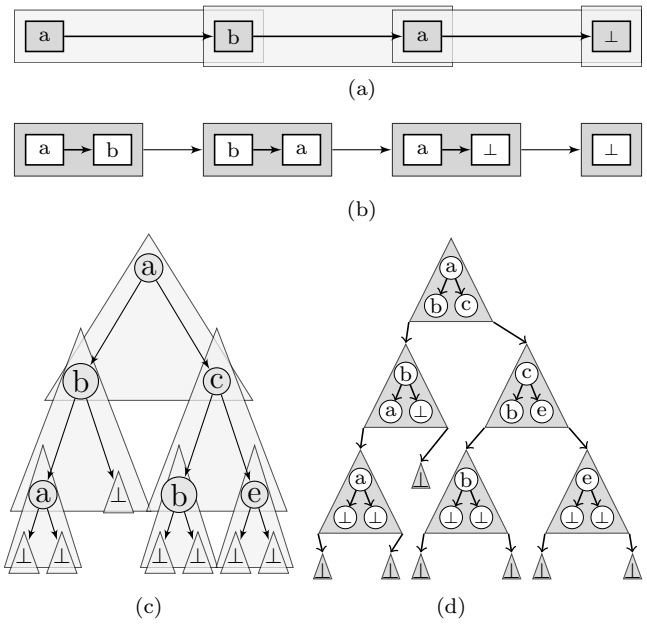


Figure 2.3: Windowed Representation. (a) List l_i, (b) Windowed List $\tilde{l}_i$, (c) Tree t_i, (d) Windowed Tree $\widetilde{T}_i$

The window-based abstraction is crucial as it allows an arbitrarily large data structures (containing possibly unbounded addresses) to be represented as a sequence of windows (with the caveat that adjacent windows that encode overlapping neighborhoods are consistent with each other). Given a graph G, we denote by $\widetilde{G}$ the corresponding "windowed" data structure. In lieu of

a formal definition, we pictorially represent this in Figure 2.3. We have not shown the local addresses in each window for simplicity.

As we will see in Chapter 3 and Chapter 4, we stipulate that methods perform localized updates to data structures. Thus, by our stipulation, each update performed by a method to a data structure is constrained within a window of some size. The motivation for the term window comes from this stipulation: Imagine the data structure as a set of customers, and the method as a clerk sitting at an office window. At any given time, only part of the set is visible to the clerk, through the window. The clerk can only take action to whichever part of the set is visible to him. Similarly, the method can only modify nodes that are part of a window.

2.4 Tree Automata

Trees. A tree is a *directed, acyclic, labeled graph* $T(V, E)$ that satisfies the property that each vertex (except the root vertex) has a single parent, and the root vertex has no parent. We say that the *branching arity* or *degree* of T is K if the maximum out-degree of any vertex in T is K. Each vertex of the tree is associated with a vertex-label defined by the function $L_V : V \mapsto \mathcal{D}$, where $\mathcal{D}$ is some finite data domain. Each edge is associated with an edge-label defined by the function $L_E : E \mapsto \{1, \ldots, K\}$. The *height* of a tree, denoted by $h(T)$, is the maximum distance of any vertex from the root vertex. We say that a tree is *finite* if $h(T)$ is a finite positive integer, and *infinite* otherwise.

Def. 2.4.1. A *nondeterministic finite state tree automaton* over an infinite

k-ary tree is a tuple $\mathcal{A} = (\Sigma, Q, \delta, q_0, \Phi)$ where:

Σ is the finite, nonempty input alphabet labeling the nodes of the tree,

Q is the finite, nonempty set of states of the automaton,

$\delta : Q \times \Sigma \mapsto 2^{Q^k}$ is the nondeterministic transition function,

$q_0 \in Q$ is the start state of the automaton, and

Φ is an acceptance condition.

The run ρ of $\mathcal{A}$ on a Σ-labeled T is an annotation of T with the states Q compatible with δ. The acceptance of T by $\mathcal{A}$ is defined by a suitable temporal property over the paths in the tree. We identify certain useful acceptance conditions that we require in this dissertation.

The *Büchi* acceptance condition requires that some state in a given set of states F occurs *infinitely often* along every path in ρ. An automaton $\mathcal{A}$ with the Büchi acceptance condition is also called a Büchi tree automaton.

The *parity* acceptance condition is specified in terms of a set of mutually disjoint subsets $\{\Phi_0, \ldots, \Phi_m\}$ of Q. If $\pi = \langle q_0, \ldots, q_i, \ldots \rangle$ is a finite or infinite sequence of automaton states, then we say that π satisfies the parity condition if the following condition is satisfied: there exists an even number r, $0 < r < m$, such that some state in Φ_r appears infinitely often in π, and each of the states in the set $\bigcup_{r<j\leq m} \Phi_j$ appears only finitely often in π. The parity condition is often alternately expressed as follows: A sequence of states π satisfies the parity condition, when the states of the automaton are colored with a set of colors $\{c_0, \ldots, c_m\}$, and for all colors that appear infinitely often in π, the color

with the highest index has an even index. We say that a run satisfies parity acceptance, if the parity condition holds along *all* paths in ρ.

Run of a Tree Automaton on an Arbitrary Graph. A tree automaton can be meaningfully defined to run on general graphs. For simplicity, consider a binary graph (graph with degree 2) We note that an infinite binary tree T can be viewed as a binary structure $M = (S, R, L)$, where $S = \{0,1\}^*$, $R = R_0 \cup R_1$ with $R_0 = \{(s, s_0) : s \in S\}$ and $R_1 = \{(s, s_1) : s \in S\}$, and $L = T$. We can extend the notion of a run of a tree automaton to appropriately labeled binary, directed graphs that are not binary trees. Such graphs, if accepted, are witnesses to the nonemptiness of tree automata. A binary structure $M = (S, R_0, R_1, L)$ consists of a state set S and labeling L as before, plus a transition relation $R_0 \cup R_1$ decomposed into two partial functions: $R_0 : S \mapsto S$, where $R_0(s)$, when defined, specifies the 0-successor of s, and $R_1 : S \mapsto S$, where $R_1(s)$, when defined, specifies the 1-successor of s. We say that M is a full binary structure iff R_0 and R_1 are total. A run ρ of automaton $\mathcal{A}$ on binary structure $M = (S, R_0, R_1, L)$, if it exists, is a mapping $\rho : S \mapsto Q$ such that for all $s \in S$, $(\rho(R_0(s)), \rho(R_1(s))) \in \delta(\rho(s), L(s))$, and $\rho(s_0) = q_0$. It turns out that if an automaton accepts some binary tree, there does exist some finite binary graph on which there is a run that is accepting: all of the paths through the graph denote state sequences of the automaton meeting its acceptance condition, [49].

Language, Product, Emptiness. Given two automata $\mathcal{A}_1 = (\Sigma_1, Q_1, \delta_2, q_{01}, \Phi_1)$ and $\mathcal{A}_2 = (\Sigma_2, Q_2, \delta_2, q_{02}, \Phi_2)$, the *synchronous product* $A_p = \mathcal{A}_1 \otimes \mathcal{A}_2$ is defined if $\Sigma_1 = \Sigma_2$. The tuple-components of A_p are defined in in terms of $\mathcal{A}_1$ and $\mathcal{A}_2$ as follows: $\Sigma_p = \Sigma_1 = \Sigma_2$, $Q_p = Q_1 \times Q_2$, $q_{0_p} = (q_{01}, q_{02})$. Each element in $\delta(q_1, \sigma)$ is a next state tuple of the form $(r_1, \ldots, r_K)$; we denote such a next state tuple by $\bar{r}$. We define a product-tuple $\bar{r} \times \bar{s}$ as the K-tuple of the form $(r_1, s_1), (r_2, s_2), \ldots, (r_K, s_K)$. Now let $q_p = (q_1, q_2)$ be a state of $\mathcal{A}_p$. Let $\delta_1(q_1, \sigma) = \{\bar{r}^1, \ldots, \bar{r}^m\}$. Similarly, let $\delta_2(q_2, \sigma) = \{\bar{s}^1, \ldots, \bar{s}^n\}$. Then $\delta(q_p, \sigma)$ is defined as the set $\{\bar{r}^i \times \bar{s}^j | 1 \leq i \leq m, 1 \leq j \leq n\}$. The acceptance condition Φ_p depends on the kind of acceptance condition used for the constituent automata. For instance, if $\mathcal{A}_1$ and $\mathcal{A}_2$ are Büchi tree automata, then Φ_p is specified in terms of the set $F_p = F_1 \cap F_2$. For parity automata, the acceptance condition for the product is defined in terms of the coloring function for the constituent automata; for more details see [25].

The language of an automaton $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set of all trees accepted by $\mathcal{A}$. We say that $\mathcal{L}(\mathcal{A})$ is *nonempty* if there exists some tree that is accepted by $\mathcal{A}$, and *empty* otherwise.

Checking Nonemptiness of a Tree Automaton. The transition function δ of a tree automaton $\mathcal{A}$ can be viewed as a bipartite AND/OR-graph, where the set of states Q comprises the set of OR-nodes, and the set of symbols Σ comprises the set of AND-nodes. Intuitively, the OR-nodes indicate a nondeterministic choice for $\mathcal{A}$, while the AND-nodes force the automaton

along all directions corresponding to their successors. For instance, consider the following set of transitions for an automaton $\mathcal{A}$ operating on a binary tree:

$$\delta(q_0, a) = \{(q_1, q_2), (q_2, q_1)\}$$
$$\delta(q_0, b) = \{(q_2, q_2)\}$$

These transitions can be represented using the AND/OR-graph shown in Figure 2.4.

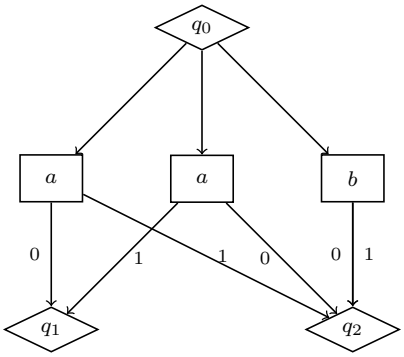


Figure 2.4: AND/OR-graph

For purpose of testing nonemptiness, without loss of generality, we can restrict our attention to a tree automaton with a single symbol alphabet. In other words, replace the symbols at the AND-nodes with the same distinguished symbol, say c. Let the resulting automaton be denoted by $\mathcal{A}'$. It can be shown that the original automaton $\mathcal{A}$ is empty iff $\mathcal{A}'$ is empty.

Henceforth, we assume that the tree automaton transition diagram is specified as an AND/OR-graph, and the input symbol alphabet consists of a single symbol. If an automaton $\mathcal{A}$ is nonempty, there is a structure contained within the AND-OR graph that represents an accepting run of $\mathcal{A}$. The root of such a structure is the OR-node corresponding to the initial state. For each OR-node o in the structure, the structure contains at least one AND-node successor of o, and at each AND-node a, the tree contains all the (OR-node) successors of a. In fact, if we restrict the structure to just its constituent AND-nodes (which is a Σ-labeled tree T), and label each AND-node with the preceding OR-node in the structure, we get precisely an accepting run of $\mathcal{A}$ on T.

We note that acceptance conditions are specified as temporal logic formulas on the sequences of states (*i.e.*, OR-nodes). Thus, if there is an accepting run of $\mathcal{A}$ on some tree T, there is a structure contained within the AND/OR-graph such that the OR-nodes of the structure satisfy the desired temporal logic formula. We can exploit this fact to check for nonemptiness. To check if $\mathcal{A}$ accepts any tree, we simply model check the AND/OR-graph representation of $\mathcal{A}$ with respect to a modified temporal logic formula denoting the acceptance condition. (The modification accounts for the extra AND-nodes between consecutive OR-nodes). For instance, the Büchi condition specifies a set of states colored *green*, and $\mathcal{A}$ accepts a tree T if it sees a *green* state infinitely often along every path in T. Thus, to check for nonemptiness of a Büchi-tree automaton, we model check the AND/OR-graph with respect to

the μ-calculus formula: $\nu Z.\mu Y.EXAX(green \vee Y \wedge Z)$. Details can be found in [56, 54, 51].

Chapter 3

Verifying Iterative Methods

Outline. In this chapter, we present an automata-theoretic framework for verification of iterative methods that modify directed graphs. We first present a bird's eye view of our solution strategy. We then discuss certain requirements that methods should satisfy to be verifiable using our approach. We then present a programming language to facilitate specification of methods that satisfy our stipulations, and present the necessary algorithms to translate a method to an equivalent method automaton. We discuss specifications provided as tree automata, and conclude with complexity analysis and related work.

3.1 Verification with Automata

We can cast the problem outlined in Section 2.2 as an equivalent problem in automata theory. Recall from Section 2.2 that we wish to verify a method $\mathcal{M}$ given a pre-condition φ and a post-condition ψ. Suppose the pre/post-conditions can be specified as automata $\mathcal{A}_\varphi$ and $\mathcal{A}_{\neg\psi}$ respectively. The pre-condition automaton $\mathcal{A}_\varphi$ accepts a graph G iff $G \models \varphi$, similarly, the post-condition automaton $\mathcal{A}_{\neg\psi}$ accepts a graph G iff $G \not\models \psi$. We can rep-

resent a method $\mathcal{M}$ by an automaton known as the *method automaton* $\mathcal{A}_{\mathcal{M}}$. Informally, $\mathcal{A}_{\mathcal{M}}$ accepts a pair of graphs (G_i, G_o) iff $G_o = \mathcal{M}(G_i)$. Consider the product $\mathcal{A}_p$ of $\mathcal{A}_{\mathcal{M}}$, $\mathcal{A}_{\varphi}$ and $\mathcal{A}_{\neg\psi}$. $\mathcal{A}_p$ is nonempty iff there exists an input graph G_i satisfying φ for which the action of $\mathcal{M}$ produces an output graph G_o that does not satisfy ψ. In effect, we can reduce checking correctness of $\mathcal{M}$ to checking the language emptiness for $\mathcal{A}_p$. Note that with sufficiently expressive automata models, the correctness problem for *any* method operating on an input graph and transforming it to an output graph can be similarly cast in terms of automata. However, except for specific methods and properties, the parameterized correctness problem is undecidable.

Theorem 3.1.1. *Given arbitrary properties φ, ψ, and an arbitrary method $\mathcal{M}$, the problem of determining if $\langle\varphi(G_i)\rangle\mathcal{M}\langle\psi(G_o)\rangle$ is undecidable.*

Proof. The proof follows from a simple reduction to the halting problem for Turing machines. A method operating on a data structure can be interpreted as a Turing Machine $TM_{\mathcal{M}}$, where the data structure represents the tape, and the finite number of statements in the method represent the finite state control of the Turing Machine. Thus, in the most general case, parameterized correctness of $\mathcal{M}$ is equivalent to checking if the initial tape configuration of $TM_{\mathcal{M}}$ satisfies φ, and if the tape configuration obtained by the actions of $TM_{\mathcal{M}}$ satisfies ψ when $TM_{\mathcal{M}}$ halts. Clearly, as the halting problem for Turing Machines is undecidable, the parameterized verification problem is undecidable. □

On the other hand, if we restrict attention to decidable fragments of

the parameterized correctness problem, we can see that it hinges on having a product automaton $\mathcal{A}_p$ with decidable nonemptiness. This is possible if: (a) each of $\mathcal{A}_{\mathcal{M}}$, $\mathcal{A}_{\varphi}$, and $\mathcal{A}_{\neg\psi}$ have a decidable emptiness problem, (b) the product operator $\otimes$ for combining these automata is well-defined, and (c) the intersection of the languages of $\mathcal{A}_{\mathcal{M}}$, $\mathcal{A}_{\varphi}$ and $\mathcal{A}_{\neg\psi}$ has decidable nonemptiness.

In this work, we restrict our attention to properties ($\mathcal{A}_{\neg\psi}$ and $\mathcal{A}_{\varphi}$) specifiable as finite state nondeterministic tree automata with suitable acceptance conditions. Thus, $\mathcal{A}_{\varphi}$ and $\mathcal{A}_{\neg\psi}$ thus trivially satisfy the requirement in (a) and (b). The action of an arbitrary $\mathcal{M}$ is equivalent to that of an arbitrary Turing Machine. However, in practice, methods can be mimicked by automata that are much simpler than Turing Machines, for instance, *nfa*s, *pda*s, finite state tree automata, pushdown tree automata, *etc.* In what follows, we identify conditions on methods that allow methods to be mimicked by finite state tree automata.

3.2 Scope

In this section, we outline the requirements that the methods verifiable in our framework should satisfy. We begin by defining a a method automaton $\mathcal{A}_{\mathcal{M}}$ as an exact abstraction for a method $\mathcal{M}$. We formalize the definition of $\mathcal{A}_{\mathcal{M}}$ below.

Def. 3.2.1 (Mimics)**.** We say that $\mathcal{A}_{\mathcal{M}}$ *mimics* a method $\mathcal{M}$ (denoted $\mathcal{A} \bowtie \mathcal{M}$), if for all input graphs G_i, $\mathcal{A}_{\mathcal{M}}$ accepts (G_i, G_o) if and only if $G_o = \mathcal{M}(G_i)$.

If a method $\mathcal{M}$ performs only a fixed amount of work (independent of the size of the data structure) on each node of its input data structure, then we show in Lemma 3.2.1 that there exists a finite state automaton $\mathcal{A}_{\mathcal{M}}$, s.t. $\mathcal{A}_{\mathcal{M}} \bowtie \mathcal{M}$. To precisely explain the notion of bounded work, we first define a *destructive update*.

Def. 3.2.2. A *destructive update* (*du*) is a modification to a data structure node n (directly by accessing n or indirectly through a pointer p to n). Let x be some element of $\mathcal{D}$, and let p_i's be pointers, then *du* is of one of the following forms:

- $n.d = x$ or $p->d = x$ (changing the data value of n),
- $n.l_i = p_j$ or $p->l_i = p_j$ (changing a link of n),
- $delete(p)$ (marking the node pointed to by p as free),
- $p->l_j = new(x, p_1, \ldots, p_K)$ (inserting a new node as a successor of node pointed to by p).

Note that a destructive update itself may be composed of multiple smaller, atomic operations, but all the operations performed by $\mathcal{M}$ once it visits a node are together termed a destructive update. We say that a method performs a destructive pass over a graph if it visits nodes in the graph starting from the root to the leaf nodes, possibly performing a destructive update at each node in the graph. For instance, a method that reverses a linked list by

setting the next pointer of every node to point to the previous node, performs a single destructive pass over the list (in which it performs one destructive update to every node). On the other hand, a method that sorts a linked list using bubble sort potentially performs k^2 destructive passes over the list.

We remark that a method that performs a bounded number of destructive passes effectively does a bounded number of destructive updates per node of the data structure. We use these two notions of "bounded work" interchangeably.

Before we formally state Lemma 3.2.1, we introduce some terminology to encode actions of methods that perform a bounded number of destructive passes. Note that any graph G can be unwound and unfolded into a (possibly infinite) tree $T(G)$. Let G_i and G_o be two arbitrary graphs of the same arity K. The *composite graph* $G_c = G_i \circ G_o$ is the (tree) graph obtained by super-imposing $T(G_i)$ and $T(G_o)$. Informally, super-imposition pairs nodes that correspond to the same topological rank in the tree, and if G_i (resp. G_o) does not have a node at a particular topological index we replace that node by $\perp$. We define projection operators $\Gamma_i(G_c)$ and $\Gamma_o(G_c)$ to obtain the graphs G_i and G_o respectively.

The composite graph can be thought of as a joint encoding of the input and output graphs for a method that performs a single destructive pass over the input graph. For instance, consider the following annotation scheme: each vertex $v \in V_c$ and each edge $e \in E_c$ annotated with one of three colors *black, green* or *red*. The color *black* represents part of the input graph that remains

the same, color *red* represents deleted nodes or edges, and the color *green* represents new nodes or edges. Also, the data values of each node of the composite graph model the old and the new data values at the corresponding node in the data structure. It is easy to see that such a G_c is a well-defined super-imposition of the input/output graphs G_i and G_o.

Lemma 3.2.1. *Let the maximum number of destructive updates performed by* $\mathcal{M}$ *on any node of the input data structure* D_i*, before it terminates, be* r*. If* $r \leq c$ *for some constant* c*, then there exists a finite state automaton* $\mathcal{A}_{\mathcal{M}}$*, such that* $\mathcal{A}_{\mathcal{M}} \bowtie \mathcal{M}$*.*

Proof. For simplicity, consider the case where $r = 1$. Let $G_c = G_i \circ G_o$. Each vertex n_c in G_c is a pair of nodes (n_i, n_o). We can now define the method automaton[1] $\mathcal{A}_{\mathcal{M}}$ to run on G_c, starting at the root of G_c in state q_0. We can view the single destructive update performed by $\mathcal{M}$ on n_i as a function (f_{du}) that maps each node n_i to some output node. $\mathcal{A}_{\mathcal{M}}$ transitions to a reject state *rej* if $n_o \neq f_{du}(n_i)$, otherwise it transitions to state q_0 along all successors of n_i. Finally, if $\mathcal{A}_{\mathcal{M}}$ reaches a terminal vertex in G_c, it transitions to *acc*. By structural induction over the nodes in the tree, for all input trees G_i for which $G_o = \mathcal{M}(G_i)$, $\mathcal{A}_{\mathcal{M}}$ accepts the composite tree $G_c = G_i \circ G_o$. If $r > 1$, *i.e.*, if $\mathcal{M}$ performs a bounded number of destructive passes over the input graph G_i, then the composite graph G_c is defined to have $r + 1$ layers, *i.e.*, $G_c = G^0 \circ G^1 \circ G^r$, where $G^0 = G_i$ and $G^r = G_o$, and $\mathcal{A}_{\mathcal{M}}$ accepts G_c if for each

[1]For further insight into obtaining a suitable $\mathcal{A}_{\mathcal{M}}$, please see Section 3.4.1.

pair (G^j, G^{j+1}), $G^{j+1} = f_{du_j}(G^j)$. By construction, if r is a constant, we can always statically define a *finite state* $\mathcal{A}_{\mathcal{M}}$ that mimics $\mathcal{M}$ and runs on such a G_c. Thus, as long as $\mathcal{M}$ performs a bounded number (r) of destructive updates to each node in G_i, there always exists an $\mathcal{A}_{\mathcal{M}}$ such that $\mathcal{A}_{\mathcal{M}} \bowtie \mathcal{M}$. □

We refer to Lemma 3.2.1 as the *bounded updates property*. Unfortunately, Lemma 3.2.2 shows that it is impossible to determine whether an arbitrary method satisfies the bounded updates property. The proof is based on a reduction from Rice's theorem [137].

Lemma 3.2.2. *For an arbitrary method $\mathcal{M}$, operating on an arbitrary graph, the question whether $\mathcal{M}$ has the bounded updates property is undecidable.*

Proof. It follows from Rice's theorem that it is undecidable if the language accepted by an arbitrary Turing Machine is regular. By Lemma 3.2.1 we know that if $\mathcal{M}$ has the bounded updates property, there is a finite state automaton $\mathcal{A}_{\mathcal{M}}$ (with the corresponding regular language $\mathcal{L}(\mathcal{A}_{\mathcal{M}})$) that mimics $\mathcal{M}$. The automaton that mimics an arbitrary $\mathcal{M}$ is an arbitrary Turing Machine. Thus, if it were decidable to check if an arbitrary $\mathcal{M}$ has the bounded updates property, it would be decidable to check if the language of the Turing Machine that mimics $\mathcal{M}$ is regular. □

3.2.1 Stipulations

We now outline the stipulations for methods to be verifiable in our framework.

Bounded Updates Stipulation. As seen in the previous section, we can mimic a method by a finite state automaton if it satisfies the bounded updates property. Henceforth, we assume that all methods satisfy this property. While Lemma 3.2.1 shows that for any $\mathcal{M}$ that has the bounded updates property, there is *some* $\mathcal{A}_{\mathcal{M}}$ s.t. $\mathcal{A}_{\mathcal{M}} \bowtie \mathcal{M}$, it does not provide a recipe for extracting $\mathcal{A}_{\mathcal{M}}$ from $\mathcal{M}$. Also, Lemma 3.2.2 establishes that trying to compile an $\mathcal{A}_{\mathcal{M}}$ from an arbitrary $\mathcal{M}$ is also futile. However, it is possible to identify a syntactic fragment such that for any method $\mathcal{M}$ in this fragment, the corresponding $\mathcal{A}_{\mathcal{M}}$ can be mechanically obtained from $\mathcal{M}$. In order to obtain such a syntactic fragment, we need further stipulations.

Localized Updates Stipulation. Most methods operating on data structures use a cursor or an iterator to traverse the data structure. Methods which have multiple cursors mimic the operation of multi-head automata. Unfortunately, the parameterized correctness problem for such methods is undecidable, since the nonemptiness problem of a k-head automaton with $k \geq 2$ is undecidable [129, 113]. Thus, we focus on methods which can be simulated by a single head automaton. Such methods can have multiple cursors, which are constrained to remain within some bounded distance at all times. In terms of syntactic constructs, absence multiple (unbounded) iterators means that we disallow *nested loops* in which the inner loops can access pointer variables initialized in the outer loop, and *global pointer variables*.

In effect, every action of the method at a particular node is localized to

a neighborhood around that node. Thus, by this stipulation, the method can be viewed as a sequence of localized updates to the nodes of G_i. In other words, we can replace the original graph G_i by its windowed version $\widetilde{G}_i$, and define $\mathcal{M}$ to operate on $\widetilde{G}_i$ to obtain corresponding $\widetilde{G}_o$, without loss of precision.

Termination Stipulation. It is stipulated that the method under consideration terminates[2]. While the bounded updates property ensures that the *observational* behavior of $\mathcal{M}$ in terms of its effect on G_i can be mimicked by a finite state $\mathcal{A}_{\mathcal{M}}$, it says nothing about the *operational* behavior of $\mathcal{M}$. Thus, it is likely that $\mathcal{M}$ performs a bounded number of updates and then gets stuck in an infinite loop. For certain restricted methods (for instance, methods on acyclic data structures), it may be possible to decide termination. However, for methods on general graphs (which may contain cycles), termination may be crucial to ensure the bounded updates property. Hence, for simplicity, we assume a proof of termination.

Finiteness of the Data Domain. We assume that the domain $\mathcal{D}$ of data values is finite. In practice, nodes can have multiple data fields. We assume that these are combined into a single data field that is a tuple of the actual data fields. If a data field has a domain that is potentially infinite (for instance, strings of arbitrary lengths), then we assume that a suitable abstraction func-

[2] A similar assumption on program termination can be found in techniques such as shape analysis [101], PALE [108], and separation logic [115], which implicitly assume the termination of the program being analyzed.

tion is provided which maps the large (possibly infinite) data domain into a small finite data domain. Since we are largely interested in structural properties of methods, such an abstraction typically does not lose precision. Note that such an abstraction is an instance of the *data independence* argument [149].

In this chapter, we assume that methods are *iterative* in nature, *i.e.*, they make of loops to traverse and possibly mutate the data structure. We assume that all calls to other methods are inlined in the method body.

3.3 Bounded Updates Programming Language

Bounded **U**p**D**ates **P**rogramming **L**anguage (Bud-Pl) is a fragment that allows methods to be written so that they satisfy the bounded updates property if they terminate. We first present the syntax for this fragment, and then present an algorithm to translate a $\mathcal{M}$ written in Bud-Pl to the corresponding $\mathcal{A}_{\mathcal{M}}$.

Table 3.1 shows the syntax for a method in Bud-Pl. In this table, we use the convention that the italicized text corresponds to the non-terminals in the grammar, while the text in the typewriter font corresponds to terminals (or keywords) in the grammar. Recall from Section 3.2.1, that we allow only a single primary pointer variable or *iterator* and a finite number of *virtual iterators* (that are always constrained to be within a bounded distance from the iterator) for a method in our syntax. For simplicity, we omit the extension to include virtual iterators. We use the keyword `iter` to denote an iterator. We

Table 3.1: BUD-PL Syntax for Iterative Methods

`Method Declaration`		
$\mathcal{M}$	::=	*sig* { *block* }
sig	::=	*name* `(iter)` | *name* `(iter,`$x_1, \ldots, x_n$`)`
`Sequential Composition`		
block	::=	*block stmt* | *stmt*
stmt	::=	*loop* | *ifthen* | *ifelse*
		| *du* | `skip` | `return`
`Conditional Statement`		
ifthen	::=	`if (`*expr*`)` { *block* }
ifelse	::=	*ifthen* `else` { *block* }
`Assignment Statement`		
du	::=	`delete` *lptr* | *lptr*`->data :=` *newdata*
		| *lptr*`:=` *ptr* | *lptr*`:=` *newnode*
ptr	::=	*lptr* | `prev`
lptr	::=	`iter` | `iter->`l_{i_1} `->`$\ldots$ `->`l_{i_z}
newnode	::=	`new node` { *d*, *ptr*, ..., *ptr* }
expr	::=	`(`*expr*`)` | *expr* `and` *expr* | *expr* `or` *expr*
		| `not` *expr* | *ptr*`->data==`*d* | *ptr* `==null`
		| *ptr* `==`*ptr*
`Loop Statement`		
loop	::=	`while (`*expr*`)` { *lbody advance* }
lbody	::=	*lbody lstmt* | *lstmt*
lstmt	::=	*du* | *lifthen* | *lifelse*
		| `skip` | `break` | `return`
advance	::=	$advance_0$ ... $advance_K$
$advance_j$	::=	`skip` | `iter:=`*lptr* `when` *expr*
lifthen	::=	`if (`*expr*`)` { *lbody* }
lifelse	::=	*lifthen* `else` { *lbody* }

use a `C`-like syntax for describing access to fields of a node. Thus, the abbreviation `iter->`f indicates the field f of the node being pointed to by `iter`. A method consists of a signature (*sig*) that defines the data arguments along with the single pointer argument and a method body (that is a block statement, *block*). The statement *block* is a sequential composition of one or more statements (denoted as *stmt*). Each *stmt* is either a conditional statement, an assignment (i.e. a destructive update) or a loop statement. A conditional statement has an associated test-expression *expr*, which is any Boolean-valued predicate that allows Boolean-valued predicates on $\mathcal{D}$ (the data domain), comparisons of the data in nodes with values from $\mathcal{D}$ and comparisons of pointer (*ptr*) values. A loop statement consists of a loop guard, (similar to *expr*), and a loop body. Similar to a *block* statement, the loop body (*lbody*) is a sequential composition of *lstmt* statements, with the exception that it cannot have nested loop statements. The only different between the conditional statements *ifthen* and *lifthen* (and the corr. statements with the `else` part) is that *lifthen* cannot have nested loop statements in it. This is to avoid getting a nested loop in an indirect fashion.

A loop statement has standard semantics, with a few additional caveats. Every update to a `iter` is preceded by storing the current value of the `iter` in a special variable called `prev`, which cannot be used on the left hand side of an assignment statement. The addition of `prev` enhances the expressive power of our language by allowing methods to perform operations based on past value of the `iter`.

A loop has a special *advance* statement that advances `iter` to the next desired node in the data structure. While we do not show this in the syntax for simplicity, care is taken to ensure that `iter` is advanced only to a node within the original input window. When a new node is created, the pointer fields of the new node are initialized to any value within the current input window. These two constructs together prevent the method from inserting an unbounded sequence of nodes by repeatedly advancing to a newly inserted node, and inserting a new node as its child. A `break` statement causes the method to break out of the loop. We note that advancing the iterator is also a guarded command, specified by the `when` keyword.

Recall that destructive updates are allowed only within the bounded window defined by the `iter`. This is ensured by restricting the left hand sides of assignments to pointer expressions (*lptr*), which only allow access to nodes at a bounded distance from `iter`. The required size of the window can be gauged by inspecting the longest such pointer expression in the method. Finally, a `skip` statement is represents an empty method action, or a `nop` instruction.

3.4 Method Automata

In this section, we present the algorithms for compiling methods into equivalent method automata.

3.4.1 Method Automaton

As discussed in the previous section, the action of an arbitrary method $\mathcal{M}$ on a data structure can be mimicked by a Turing Machine. However, with the stipulations in Section 3.2.1, methods are substantially simpler, and can, in fact, be mimicked by a finite state transducer. A transducer is an acceptor with an additional output language, and rules to specify an output symbol corresponding to each state and an input symbol. We choose to model methods as accepting automata, known as *method automata*, instead of transducers, in order to streamline the product construction with the pre/post-condition automata, and make use of efficient algorithms for emptiness and intersection for traditional automata.

The proof of Lemma 3.2.1 hints at a constructive approach to obtain a finite state $\mathcal{A}_{\mathcal{M}}$ that mimics a method $\mathcal{M}$. Here, we refine this idea and present an algorithm to do so. By Def. 3.2.1, we know that a method automaton $\mathcal{A}_{\mathcal{M}}$ accepts a pair of graphs (G_i, G_o) iff $G_o = \mathcal{M}(G_i)$. It is clear that when reading graphs G_i and G_o, $\mathcal{A}_{\mathcal{M}}$ would need to incorporate some mechanism to check if G_o is the same as $\mathcal{M}(G_i)$. As the graphs G_i and G_o could be arbitrarily long, in order to make this check, $\mathcal{A}_{\mathcal{M}}$ would have to remember an arbitrarily large amount of information if it reads G_i and G_o independently. Needless to say, $\mathcal{A}_{\mathcal{M}}$ would fail to be finite state. In order to enable construction of a finite state $\mathcal{A}_{\mathcal{M}}$, we define $\mathcal{A}_{\mathcal{M}}$ to run over a suitable composite graph.

From the localized updates stipulation in Section 3.2.1, we know that $\mathcal{M}$ can be viewed as a sequence of destructive updates to $\widetilde{G}_i$, *i.e.*, the windowed

version of the input graph G_i. Let $\widetilde{G}_i$ be some windowed input graph, and $\widetilde{G}_o$ be a possible output graph, and let $\widetilde{G}_c = \widetilde{G}_i \circ \widetilde{G}_o$ be the composite graph obtained by super-imposing $\widetilde{G}_i$ and $\widetilde{G}_o$. Each node of $\widetilde{G}_c$ is thus a pair of windows (w_i, w_o), where w_i is a vertex in $\widetilde{G}_i$ and w_o is a vertex in $\widetilde{G}_o$. The $\widetilde{G}_o$ component of $\widetilde{G}_c$ represents a possible guess for an execution of $\mathcal{M}$ on (the G_i corresponding to) $\widetilde{G}_i$. $\mathcal{A}_{\mathcal{M}}$ checks if this guess is correct, by iteratively checking if the guessed value of the output window matches the action of $\mathcal{M}$ on the input window. Thus, an each atomic step of $\mathcal{A}_{\mathcal{M}}$ needs to be able to model a destructive update. We show how we can achieve this below.

Modeling destructive updates. A destructive update du to a node n, either marks a node n as deleted, or changes $n.d$ or $n.l_i$ (for some i). We can view du as a function mapping an "input" window w_i to an "output" window w_o, where w_o is obtained by performing the actions of du on w_i. Thus for statement: $p->d{:=}x$, w_o is identical to w_i except for $p->d$, which has the value x. When du modifies $p->l_i$, the expression on the RHS of du is a pointer expression, or a new node. The effect of statement $p->l_i{:=}p_j$, is to set $p->l_i$ in w_o to $laddr(p_j)$. Insertion of a new node is modeled by adding a new node to w_o and setting $p->l_i$ to the $laddr$ of this new node. Deleting a node is modeled by over-writing each field in the corresponding node in w_o with some special character (say '$-$'). Thus, for any destructive update du, we can compute the map from w_i to w_o, denoted by the function f_{du}.

3.4.2 Translation Algorithm

The method automaton $\mathcal{A}_{\mathcal{M}}$ is the tuple $(\Sigma, Q_{\mathcal{M}}, \delta_{\mathcal{M}}, q_{0,\mathcal{M}}, \Phi_{\mathcal{M}})$ where each component of the tuple has the standard meaning. For a method $\mathcal{M}$ that operates on graphs with degree K, $\mathcal{A}_{\mathcal{M}}$ is a K-ary tree automaton. We assume that each statement in $\mathcal{M}$ is labeled with a unique line number $\{1, \ldots, |\mathcal{M}|\}$, where $|\mathcal{M}|$ is the length of $\mathcal{M}$. The parity acceptance condition $\Phi_{\mathcal{M}}$ is specified using two colors $\{(red = c_1), (green = c_2)\}$. States colored *green* are accepting states and those colored *red* are rejecting.

Recall from Section 2.3, the window-based abstraction for graphs. From Table 3.1, we can see that each destructive update is restricted to a set of nodes that are a bounded distance from the special pointer variable `iter`. Thus, each destructive update is restricted to a window. Each node of the (windowed) composite graph $\widetilde{G}_c = \widetilde{G}_i \circ \widetilde{G}_o$ is a pair of windows (w_i, w_o).

Consider a composite graph in which a node $w_c = (w_i, w_o)$ is followed by nodes $x1_c = (x1_i, x1_o), \ldots, xK_c = (xK_i, xK_o)$ corresponding to the K children of w_c. Suppose $\mathcal{A}_{\mathcal{M}}$ is reading the node w_c, and will advance to the nodes $x1_c, \ldots, xK_c$. The actions of $\mathcal{A}_{\mathcal{M}}$ are as follows:

1. *Remembering destructive updates:*

   $\mathcal{M}$ possibly destructively updates w_i multiple times before finally "leaving" w_i. Thus, when at a particular node, $\mathcal{A}_{\mathcal{M}}$ stores the most recent value of the input window in its state.

2. *Checking if G_c encodes action of $\mathcal{M}$:*

   Before advancing to the next symbol xj_c (by setting `iter` to `iter`$->l_j$) $\mathcal{A}_{\mathcal{M}}$ checks if the action of the sequence of destructive updates performed by $\mathcal{M}$ on w_i (which is remembered in the state of $\mathcal{A}_{\mathcal{M}}$) is the same as w_o, *i.e.*, $\mathcal{A}_{\mathcal{M}}$ checks if $w_o = \mathcal{M}(w_i)$.

3. *Remembering changes:*

   When $\mathcal{A}_{\mathcal{M}}$ advances to the next symbol xj_c, the overlapping nodes in the window w_i and xj_i could be possibly changed due to the action of $\mathcal{M}$. By remembering the portion of w_o that overlaps with xj_i, $\mathcal{A}_{\mathcal{M}}$ knows the most recent values of these overlapping nodes. Let the height of the windows w_i, w_o be z. We denote the overlapping portion by $succ(j, w_o)$, *i.e.*, the sub-graph rooted at the j^{th} successor of w_o. Let $tail(xj_i)$ denote the leaf nodes of of xj_i. Essentially, once $\mathcal{A}_{\mathcal{M}}$ arrives at a new symbol xj_c, due to the actions of $\mathcal{M}$ on the previous symbol (w_c), the value of xj_i is *already* modified to the window obtained by splicing together $succ(j, w_o)$ and $tail(xj_i)$. Let r_j denote the short-hand for $succ(j, w_o)$. We denote the splice by $r_j \hookrightarrow w_o$.

4. *Consistency check:*

   After reading xj_c, $\mathcal{A}_{\mathcal{M}}$ checks if the portion of the current window that overlaps with the previous input window is the same in both windows. We call this the consistency check. Let $head(xj_i)$ denote the sub-graph of

height $z-1$ rooted at the root of xj_i. Essentially, we check if $head(xj_i) = succ(j, w_i)$.

Thus, the state of $\mathcal{A}_\mathcal{M}$ is a tuple (ℓ, r_i, r_o, cw), where ℓ is the line number in $\mathcal{M}$, r_i (resp. r_o) is the trailing portion of the input (resp. output) window, and cw is the current (possibly modified) value of the input window. We describe the algorithm to populate the transition relation of $\mathcal{A}_\mathcal{M}$ in Algorithm 3.1. This algorithm implements Steps 1-4 as discussed above.

We assume that each statement s_ℓ in $\mathcal{M}$ is labeled with a unique line number ℓ. The succ (resp. prev) returns the statement after (resp. before) s_ℓ in $\mathcal{M}$, and `lineNum` returns the line number for any statement. We use reject and accept as macros to add transitions to special *reject* and *accept* states from a given state, on a given symbol. Algorithm 3.1 uses a while-loop to iterate over each statement s_ℓ in $\mathcal{M}$ (Line 29). The set Q^ℓ denotes the set of reachable states for $\mathcal{A}_\mathcal{M}$ before executing the statement s_ℓ. For each s_ℓ, the algorithm iterates over the set of states Q^ℓ and all possible input/output window pairs (w_i, w_o) and (possibly) adds transitions to $\mathcal{A}_\mathcal{M}$ to mimic the action of $\mathcal{M}$. If the predecessor of s_ℓ has advanced the iterator `iter`, then in Lines 10-9, we check if the overlapping value of the "previous" input window (remembered in the r_i portion of the state) is consistent with the current input window. If not, we add transitions to a special *reject* state from the state q on the symbol (w_i, w_o) (Line 8).

Let s_ℓ cause the iterator `iter` to advance to the j^{th} successor of the

Algorithm 3.1: CompileIterative

```
1  begin
2      ℓ :=1 , q_0 = (0, ∅, ∅, ∅), Q^1 = {q_0} ;
3      while (ℓ < |M|) do
4          foreach ( q in Q^ℓ, (w_i, w_o) in (W(z) × W(z)) ) do
5              n = lineNum(succ(s_ℓ)) ;
6              if ( prev (s_ℓ) advanced iter ) then            // check consistency
7                  if ( (r_i ≠ head(w_i)) && (ℓ ≠ 0) ) then
8                      reject (q, (w_i, w_o)) ;
9                      continue ;
10                 cw := r_o ↪ w_i ;                            // create splice
11             switch s_ℓ do
12                 case [du advances iter to iter->l_j] :
                       /* Check if G_c encodes the actions of M */
13                     if (cw ≠ w_o) then reject (q, (w_i, w_o)) ;
14                     else
15                         r_i :=succ(j, w_i), r_o :=succ(j, w_o), q' :=(n, r_i, r_o, ∅) ;
16                         if (j = k) then nextTuple[k] :=q' ;
17                         else nextTuple[k] :=accept;
18                         addTransition(q, (w_i, w_o), nextTuple) ;
19                         Q^n :=Q^n ∪ {q'} ;
20                 case [du does not modify 'iter' ] :
                       /* Memorize destructive updates */
21                     if (cw = ∅) then cw:=du(w_i) ;          // at the root node
22                     else cw:= du(cw) ;
23                     Q^n := Q^n ∪ {(n, r, cw)} ;
24                 case [if (expr) { t : s_t; ... } else { e : s_e; ... }] :
25                     if (cw ⊨ expr) then Q^t :=Q^t ∪ (t, r_i, r_o, cw) ;
26                     else Q^e :=Q^e ∪ {(e, r, cw)} ;
27                 case [ while (expr) { t : s_t; ...; }] :
28                     if (cw ⊨ expr) then Q^t :=Q^t ∪ {(t, r, cw)} ;
29                     else Q^n :=Q^n ∪ {(n, r, cw)} ;
30                 case [break]:
31                     n' :=lineNum(first statement after loop statement) ;
32                     Q^n' :=Q^n' ∪ {(n', r, cw)} ;
33                 case [advance]: similar to Line 19 ;
34                 case [halt/return]: accept (q, (w_i, w_o)) ;
35         ℓ :=ℓ + 1 ;
36 end
```

node pointed to by `iter`. Before advancing, we need to check if the "final" value of the current window (as remembered in the *cw* component of the state q) matches the output window w_o. If not, $\mathcal{A}_{\mathcal{M}}$ transitions to a reject state Line 13. If yes, we remember $r_i = succ(j, w_i)$ and $r_o = succ(j, w_o)$ in the state q' (Line 16). $\mathcal{A}_{\mathcal{M}}$ then transitions to the state q' along the j^{th} successor, and trivially accepts along all other successors. Further, we add state q' as a reachable state for the statement following s_ℓ.

If s_ℓ is a destructive update that does not advance the iterator, then we simply remember its effect in the *cw* component of the state (Lines 23-22). In automata-theoretic terms, this can be viewed as an ϵ-transition, which is used only to keep track of intermediate actions of $\mathcal{M}$ till it reaches a statement that advances the iterator.

Conditional and loop statements cause similar ϵ-transitions. If the current value of the input window, *cw*, satisfies the loop condition *expr*, then we add the state q as a reachable state for the first statement within the *if*-block, else we add it as a reachable state for the first statement within the *else*-block (Lines 26-26).

Loop guards are handled similar to conditional statements (Lines 29-29). The main difference is if *cw* does not satisfy the loop guard, transitions are added to the statement following the loop statement. Inside the loop body, individual statements are processed according to the their types. An exit from the loop (a `break` statement) causes state q to be placed in the reachable states for the first statement after the loop statement. If $\mathcal{M}$ reaches a halt-point due

to a `return` statement, $\mathcal{A}_{\mathcal{M}}$ transitions to the *accept* state (Line 34).

Lemma 3.4.1. *Methods that perform a bounded number of destructive passes over the input graph can be mimicked by a finite state tree automaton.*

We omit a formal proof for brevity. The basic idea is to encode the changes for each pass in the composite graph. Assuming that we make at most b destructive passes, the composite graph is represented as a $b+1$-tuple, $G_c = (G_0, G_1, \ldots, G_b)$ with $G_0 = G_i$ and $G_b = G_o$. Intuitively, the result of the m^{th} traversal is encoded as G_m and the automaton can verify that the graph $G_m = \mathcal{M}(G_{m-1})$.

Remark: The use of a Büchi or parity acceptance conditions for $\mathcal{A}_{\mathcal{M}}$ might seem ill-conceived as any input graph that $\mathcal{A}_{\mathcal{M}}$ is designed to run over is essentially finite. We resolve this by additing gratuitous self-loops to the `null` nodes in the input/output graphs. This makes it possible to the unwind the input/output graphs into infinite trees suitable for these acceptance conditions. The motivation for using these acceptance conditions lies in the fact that pre/post-condition automata may be defined with the parity or Büchi condition. Thus, for the product construction to be well-defined, we need to define $\mathcal{A}_{\mathcal{M}}$ with a similar acceptance condition.

Finally, we formalize the correctness of Algorithm 3.1 in the Theorem 3.4.1.

Theorem 3.4.1 (Correctness)**.** *$\mathcal{A}_{\mathcal{M}}$ derived using Algorithm 3.1 for a method written with syntax specified in Table 3.1, has the following properties: (a) $\mathcal{A}_{\mathcal{M}}$*

is a finite-state tree automaton, (b) $\mathcal{A}_{\mathcal{M}}$ *rejects* $\widetilde{G}_c = \widetilde{G}_i \circ \widetilde{G}_o$ *if* $\widetilde{G}_i$ *is inconsistent, and (c)* $\mathcal{A}_{\mathcal{M}}$ *accepts* $\widetilde{G}_c$ *iff* $G_o = \mathcal{M}(G_i)$, i.e., $\mathcal{A}_{\mathcal{M}} \bowtie \mathcal{M}$.

Proof. To prove that $\mathcal{M}$ compiles into a well-defined $\mathcal{A}_{\mathcal{M}}$, we can use structural induction on the method syntax: we can show that each statement causes only a finite number of states and transitions to be added to $\mathcal{A}_{\mathcal{M}}$. To show (a), *i.e.*, $\mathcal{A}_{\mathcal{M}}$ is finite state, we note that each state in $\mathcal{A}_{\mathcal{M}}$ is a tuple containing a line number, and three windows (two of height $z-1$, and one of height z, where z is the diameter of the window). Thus, the number of states of $\mathcal{A}_{\mathcal{M}}$ is finite, as there are a finite number of windows of a given diameter z. We note that the *cw* component is only used by the ϵ-transitions and can be removed during state minimization of the automaton. We note that (b) is true by construction.

We note that (b) ensures that any graph accepted by $\widetilde{G}_c$ corresponds to the windowed version of some valid input graph. We note that we can always uniquely obtain the original graph from the windowed graph by eliding the repeated nodes in overlapping portions of neighboring windows. Thus, $\mathcal{A}_{\mathcal{M}}$ accepts only those graphs $\widetilde{G}_c$ such that each node (which is a pair of input/output windows) of $\widetilde{G}_c$ encodes an action of $\mathcal{M}$. By structural induction on $\widetilde{G}_c$, we can we can show that $\mathcal{A}_{\mathcal{M}}$ accepts only those graphs that represent a superposition of the windowed versions of graphs G_i, G_o, s.t. $G_o = \mathcal{M}(G_i)$. □

3.5 Specifications

Tree automata are widely used to specify shape properties [42, 41, 19, 20]. One of the inputs to our technique is pre/post-conditions specified as nondeterministic finite tree automata. We now give a few examples of shape properties that can be specified as automata.

Acyclicity. Acyclicity in a binary graph can be checked by the following parity tree automaton $\mathcal{A}_{acyc}$:

$$
\begin{aligned}
&Q = \{q, q_f\} \\
&q_0 = q \\
&\delta(q, \sigma) = \begin{cases} (q_f, q_f) & \sigma = \texttt{null} \\ (q, q) & \text{otherwise} \end{cases} \\
&\delta(q_f, \sigma) = (q_f, q_f) \\
&colors = \{c_1 = \{q\}, c_2 = \{q_f\}\}
\end{aligned}
$$

Essentially, this automaton scans for the terminal `null` node along each path, and transitions to state q_f upon reading `null`. Once $\mathcal{A}_{acyc}$ reaches q_f, it stays in q_f for all input symbols. By the parity condition, $\mathcal{A}_{acyc}$ accepts the binary graph if along every path it sees the color c_2 infinitely often. This can happen only if each path ends in a `null` node, or the binary graph is acyclic. Note that we can define the same automaton as a Büchi automaton, with $\{q_f\}$ being the set of accepting states specified by the Büchi condition.

Existence of a cycle. Existence of a cycle in a binary graph can be checked by the following parity tree automaton $\mathcal{A}_{cyc}$:

$$
\begin{aligned}
&Q = \{q, q_r, q_f\} \\
&q_0 = q \\
&\delta(q, \sigma) = \begin{cases} \{(q_r, q_r)\} & \sigma = \texttt{null} \\ \{(q, q_f), (q_f, q)\} & \text{otherwise} \end{cases} \\
&\delta(q_r, \sigma) = \{(q_r, q_r)\} \\
&\delta(q_f, \sigma) = \{(q_f, q_f)\} \\
&colors = \{c_1 = \{q_r\}, c_2 = \{q, q_f\}\}
\end{aligned}
$$

In contrast to the automaton for acyclicity, this automaton is nondeterministic. If $\mathcal{A}_{cyc}$ reads a `null` node, it transitions to a rejecting q_r state. Otherwise, in each step $\mathcal{A}_{cyc}$ guesses a successor that is likely to be part of a cycle (by transitioning to q), and trivially accepts along the other successor (by transitioning to q_f). If the input graph has a cycle, there exists a run of $\mathcal{A}_{cyc}$ in which it continually makes the right choice staying in state q along the cycle, and trivially accepts all other paths. As q is an accepting state, $\mathcal{A}_{cyc}$ accepts the graph along this path. On the other hand, if the input graph is acyclic, $\mathcal{A}_{cyc}$ will reach a `null` node in state q along some path and transition to the reject states q_r. Since there will always be one path that is rejected, there is no run of $\mathcal{A}_{cyc}$ on an acyclic graph that is accepting.

Note that for the case when the degree of the input graph K, is 1 the automaton for accepting cycle containing graphs is considerably simpler and

can be obtained by simply taking the automaton for acyclicity, and switching the colors c_1 and c_2 on its states.

Finally, note by using the set $\{q, q_f\}$ as the set of accepting states in the Büchi condition, we can obtain a (nondeterministic) Büchi tree automaton that is equivalent to $\mathcal{A}_{cyc}$.

Reachability of a given data value. Suppose, given a binary tree we wish to determine if there exists a node with a given data value (key) reachable from the root node of the graph. Automaton $\mathcal{A}_{reach}$ described below can achieve this:

$$
\begin{aligned}
&Q = \{q, q_f\} \\
&q_0 = q \\
&\delta(q, n) = \begin{cases} \{(q_f, q_f)\} & n{-}{>}\texttt{data} == key \\ \{(q, q_f), (q_f, q)\} & \text{otherwise} \end{cases} \\
&\delta(q_f, \sigma) = \{(q_f, q_f)\} \\
&colors = \{c_1 = \{q\}, c_2 = \{q_f\}\}
\end{aligned}
$$

Intuitively, this automaton guesses a path to the node with the value key, and trivially accepts along other paths. If key is found, $\mathcal{A}_{reach}$ transits to a final state along each successor. If a node with `data` value key does not exist in the given graph, then $\mathcal{A}_{reach}$ will be in state q along some path, and hence reject the graph.

Sortedness. A linked structure satisfies the sortedness property if within each bounded window of size two, the value of the current node is smaller (or greater) than the successor node. An automaton that checks if a list is sorted in ascending order rejects the list iff there exists a window such that the data value of the current node is greater than the data value of the successor node.

In a variant approach, for some properties, we can use a suitable temporal logic in lieu of automata to specify properties. We now consider two such examples:

Reachability. Since specific nodes in the data structure are usually specified by their pointers, we are interested in checking reachability of pointer expressions, where `x` and `y` are pointers to nodes n_x and n_y respectively. We introduce virtual nodes labeled with v_x and v_y, such that $v_x{->}l_1 = n_x$ and $v_y{->}l_1 = n_y$. We then check if $\mathbf{AG}(\mathbf{EX}v_x \Rightarrow \mathbf{EFEY}v_y)$. Intuitively, this formula checks that for all nodes being pointed to by $v_x{->}l_1$ (alias for `x`), there exists some node n_y being pointed to by $v_y{->}l_1$ (alias for `y`) which is reachable from `x`[3].

Sharing. A node n in a data structure is called *shared* if there exist two distinct nodes, x and y in the graph such that they have n as the common immediate successor. We say that *sharing exists* in a graph if there exists a

[3] **EY** is a temporal operator in CTL with branching past, which means *there exists a yesterday*, where "yesterday" means the immediate previous step [97]

node in the graph which is *shared*. The following specification in CTL with branching past is satisfiable only if the graph contains sharing: $(x \equiv \mathbf{EY}(n)) \wedge (y \equiv \mathbf{EY}(n)) \wedge (x \not\equiv y)$.

Note that formulas in CTL and CTL_{bp} can be translated into equivalent tree automata at a cost that is exponential in the size of the formula [97]. Thus, as a first step we can translate the temporal logic specifications into tree automata, and use these in our technique.

As a final remark, we note that the pre/post-condition automata, as provided to our technique, may be defined over nodes of graphs. However, the method automaton $\mathcal{A}_{\mathcal{M}}$ is defined over windowed input/output graphs. Thus, we first need to convert the pre/post-condition automata from the given form to a windowed form. This can be easily done for the pre-condition automaton by adding the appropriate consistency checks to the automata. The post-condition automaton is nuanced as the output component of the composite graph may not be a consistent windowed version of the actual output graph. In this case, constructing the post-condition automaton from the non-windowed version requires some additional machinery. For details see [43].

3.6 Complexity Analysis

The complexity of testing nonemptiness of the product automaton $\mathcal{A}_p$, depends on the sizes of the pre/post-condition automata $\mathcal{A}_{\varphi}$ and $\mathcal{A}_{\neg\psi}$ and the method automaton $\mathcal{A}_{\mathcal{M}}$. A method $\mathcal{M}$ having $|\mathcal{M}|$ lines of code gives rise to an automaton of size $O(|\mathcal{M}|)$ states. However, the size of $\mathcal{A}_{\mathcal{M}}$ is dominated by the

sizes of $\mathcal{D}$, the degree of the graph K, and the window size z. The dependence of $|\mathcal{A}_{\mathcal{M}}|$ on $|\mathcal{D}|$ is polynomial, while its dependence on K and z is exponential.

The number of states of $\mathcal{A}_p$ proportional to the product of the number of states of its constituent automata. Hence the number of states of $\mathcal{A}_p$ is linear in the number of states of the property automata and the size of the method. Since the number of colors used for the parity condition by the property and method automata is fixed (and typically small), the number of colors used by the product automaton is also fixed.

The complexity of checking nonemptiness of a parity tree automaton is polynomial in the number of states [56, 54] (for a fixed number of colors in the parity acceptance condition). Thus, our solution is polynomial in the size of the method as well as the sizes of the property automata. Note that for linear graphs (such as lists) the method automaton and property automata can be specialized to automata on strings (*i.e.nfa*s and *dfa*s) and thus the complexity of our technique is *linear*.

In case the specifications are provided as Büchi-tree automata, the complexity of nonemptiness is quadratic in the number of states of $\mathcal{A}_p$ [56].

3.7 Bibliographic Notes

Techniques such as shape analysis [131], pointer assertion logic engine [108] and separation logic [45] address a similar genre of problems as the framework presented here. We make a brief comparison with these techniques in

what follows.

Shape Analysis. Shape analysis is a technique for computing shape invariants for programs by providing over-approximations of structure descriptors at each program point using 3-valued logic. Shape analysis typically uses static analyses to compute shapes. Shape analysis can be used to analyze a broad class of methods, but to our best knowledge provides approximate results in double exponential time.

Predicate abstraction has also been used for shape analysis in [6, 106, 14]. [6, 14] focus on singly linked lists, and [7] extends the authors' previous work to an abstraction-refinement approach for single-parent heaps. While [35] provides a way to combine predicate abstraction and model checking, it may require hints to converge to a solution. Bottom-up shape analysis [73] for heap-manipulating programs computes Hoare triples as summaries for a given method. It may be possible to combine our technique with bottom-up analysis by substituting method fragments that do not respect our imposed syntax rules with equivalent summaries, thereby allowing us to model a larger class of methods.

Logic-based Approaches. In [9], the authors discuss a decidable logic L_r for describing linked data structures. However, their work does not provide a practical algorithm for checking the validity of formulas in this logic and the complexity of the given decision procedure is high. In [152], the authors de-

scribe a logic of reachable patterns that is undecidable, which when restricted to certain reachability patterns, yields a decidable fragment reducible to MSO on trees that can be checked in double exponential time. The restrictions imposed to obtain decidability are incomparable to this work, and combinations of decidable theories is likely. The worst-case complexity of this logic is doubly exponential.

In [155], the authors present a verification technique for functional correctness of data structure implementations in `Java`, using the Jahob verification system with respect to specifications written in higher-order logic. The verification problem, which is inherently undecidable is split into a conjunction of sub-formulae, each of which is solved by an array of solvers that includes: first-order theorem provers, a host of decision procedures such as those for monadic-second order logic and Presburger arithmetic, and interactive theorem provers. This technique may be able to engineer a verification solution; however, it may involve substantial manual effort in writing specifications that are amenable to splitting, and in guiding the proof of correctness for verification conditions that are beyond the scope of fully automated techniques. This approach is an extension of the Hob analysis system [99].

In [94, 104, 105], the authors present an approach for automatically generating representation invariants of complex data structures, with a special focus on graph properties. The approach makes use of the relational first-order logic analyzer - Alloy [88]. The representation invariants are output as `Java` methods that operate on a given data structure, and determine if it satisfies

the invariant. Such representation invariants are highly useful as specifications, and it would be interesting to see if these could be automatically translated into tree automata, and subsequently used in our verification framework. In [153], the authors explore the use of sequential circuits for encoding first order relational logic formulas (written in the Alloy modeling language). Such an encoding can be used to show the equivalence of different logical formulas specifying a given shape property.

PALE. Pointer Assertion Logic Engine tool [108] encodes programs and partial specifications as formulas of monadic second order logic. Though their approach can handle a large number of data structures and methods, the complexity of the decision procedure is non-elementary. Moreover, the technique works only for loop-free code and loops need to be broken using user specified loop invariants.

Separation Logic. Classical separation logic [45], which is an extension of Hoare Logic for giving proofs of partial correctness of methods, has been traditionally used for manual proofs or in conjunction with a theorem prover. Classical separation logic without arithmetic is not recursively enumerable [125]. However, recent work has focussed on automation, by deriving decidable fragments for programs operating on structures with single successors [11].

Automata-based Approaches. In [20], system configurations (states) are trees succinctly encoded as tree automata. The transition relation is expressed

as a bottom-up tree transducer τ, and the technique checks if the transitive closure of τ applied to the initial states reaches a bad state. Though this is undecidable, the authors use abstraction-refinement to obtain a conservative solution. This approach known as *abstract regular tree model checking* (ARTMC), and has been extensively used for verifying properties of programs that manipulate tree-like data structures. [128] contains a comprehensive treatment of the application of ARTMC to verifying dynamic data structures. [19], a precursor to most work on ARTMC, presents a fully-automated technique for verifying programs manipulating 1-selector linked structures.

[75] considers the problem of proving the termination of programs manipulating tree-like structures. The approach, based on a counterexample-guided abstraction-refinement loop uses abstract regular tree-model checking to infer program invariants, and translates the program into a counter automaton. If the counter automaton can be shown to terminate, then the program terminates, else the counterexample is analyzed for feasibility.

[76] describes tree automata with size constraints that are used to verify methods modifying balanced trees. The paper allows reasoning about algebraic path properties such as balancedness using tree automata with size constraints. However, the complexity of the decision procedure is high.

Chapter 4

Verifying Recursive Methods

In the previous chapter, we presented an automata-theoretic framework for verifying correctness of iterative methods operating on parameterized data structures. In this chapter, we extend this framework to reasoning about recursive methods. We first look at some of the challenges posed by recursion, and then define a syntactic class of methods that can be mimicked using finite state tree automata. For expository reason, we divide this class into tail-recursive methods and methods with non-tail recursion. We conclude with a discussion on the related work.

4.1 Scope

Recursive methods can inherently "remember" a long history of computations, and are thus harder to model than iterative methods. Consider the simple case of an arbitrary recursive method $\mathcal{M}$ operating on a list of fixed size. For simplicity, consider the case where $\mathcal{M}$ inspects a single node in the list at any given time. Recursion allows $\mathcal{M}$ to change the current node being inspected, traverse to the successor node, or return back to the parent node. In effect, recursion allows $\mathcal{M}$ to move in both directions along the input, pos-

sibly modifying it. Thus notions of destructive passes over the data structure are consequently harder to define.

In effect, an arbitrary recursive method simulates the action of a *linear bounded automaton* (LBA). To check for correctness of such a $\mathcal{M}$ would require us to check the emptiness of an LBA, which is an undecidable problem [86].

We are interested in obtaining syntactic sub-classes of recursive methods for which verification is efficiently decidable. Recall from Section 3.2 that if a method $\mathcal{M}$ performs only a bounded number of destructive updates to the underlying data structure, then $\mathcal{M}$ can be mimicked by a finite state automaton $\mathcal{A}_{\mathcal{M}}$ (also known as the method automaton). We use this result (Lemma 3.2.1) as a starting point to to obtain decidable syntactic sub-classes of recursive methods by identifying methods that satisfy the bounded updates property. In this work, we focus on recursive methods that traverse the input graphs in a depth-first fashion.

Control Flow Structure of Recursive Methods. Similar to the stipulations in Section 3.2, and similar to iterative methods as specified in Section 3.3, we assume that the method $\mathcal{M}$ has a single pointer and possibly multiple data values as arguments. As before, the pointer argument is referred to as the iterator (`iter`). Let the node pointed to by `iter` be denoted $v_{\mathtt{I}}$. We say that $\mathcal{M}$ *visits* a node n pointed to by a pointer variable p when $\mathcal{M}$ is invoked with p as an argument. The control-flow structure of a recursive method $\mathcal{M}$ that traverses a graph in a depth-first fashion is as follows: $\mathcal{M}$ begins the traversal

at the *root* node. For every node $v_{\mathtt{I}}$, $\mathcal{M}$ visits the child nodes of $v_{\mathtt{I}}$ in some order, possibly interspersed with destructive updates to n. $\mathcal{M}$ returns back to the parent of $v_{\mathtt{I}}$ once it is done with all its recursive visits to the children of $v_{\mathtt{I}}$.

4.1.1 Bounded Updates Property Revisited

We first show the ways in which such recursive methods fail to satisfy the bounded updates property. As in Section 3.2.1, we stipulate that the updates are localized, and hence prevent the use of global pointer variables.

Unrestricted Visits to Child Nodes. For simplicity, consider a recursive method operating over a list. At a specific node n, let $visits(\mathcal{M})$ denote the number of times $\mathcal{M}$ visits it. Suppose the control-flow of $\mathcal{M}$ allows two recursive visits for every child node of a node. As shown in Figure 4.1, $\mathcal{M}$ would visit the root node of the list *once*, the successor of the root node *twice*, the node at distance 2 from the root *four times*, and so on. In particular, it visits a node at distance ℓ from the root 2^{ℓ} times. Clearly, if it does a destructive update, every time it visits a node, then it can perform 2^{ℓ} destructive updates for a node at distance ℓ from the root. The same argument holds for a $\mathcal{M}$ operating over a tree - the leaf node of a tree of height h would be visited $O(r^h)$ times, where the number of visits of $\mathcal{M}$ for each child node is $O(r)$. We note that meaningful methods that need an unrestricted number of visits to each child node are highly uncommon.

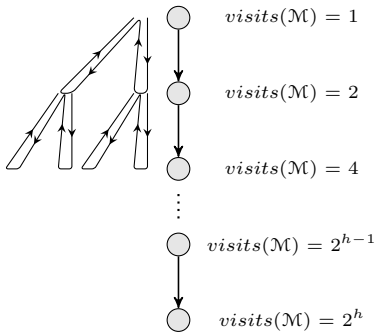


Figure 4.1: Recursive Method on a List

Sharing. Suppose we restrict the control-flow of recursive methods so that they can visit a child node of a given node at most once. However, even with this restriction, if the underlying graph contains sharing, the number of destructive updates performed by a method could be proportional to the size of the graph, *i.e.*, not bounded by a constant. We show an example of this in Figure 4.2. Here, a node at the bottom of the h^{th} diamond could be visited 2^h times.

Directed acyclic graphs (*dag*s) allow sharing of isomorphic sub-graphs in order to reduce space complexity. Hence, most methods on *dags* typically visit any given sub-graph at most once. Thus, for a large class of methods operating on *dags*, it is reasonable to assume that these methods would visit nodes in the *dags* at most once, and thus trivially satisfy the bounded updates property. We call this the *single visit property*.

While it is relatively easy to syntactically enforce the number of visits allowed per child node, it is much harder to enforce the *single visit property*

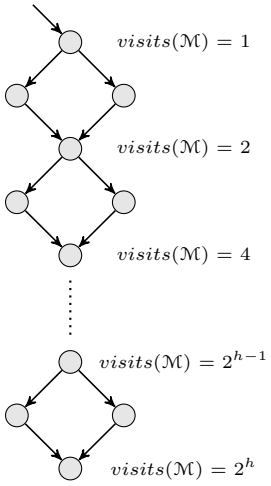


Figure 4.2: Recursive Method on a Diamond Graph

syntactically. In what follows, we thus focus on tree-like data structures, *i.e.*, data structures in which each node has a unique parent node. A decidable syntactic fragment for methods operating on *dags* is an interesting open problem. For ease of discourse, we divide the class of recursive methods on tree-like data structures into two sub-classes: tail-recursive methods, and the more general class that uses non-tail recursion. The syntax presented herein for methods operating on tree-like data structures guarantees that methods written using this syntax satisfy the bounded updates property. This helps eliminate manual effort that is required for iterative methods, where a proof of the bounded updates property is necessary step before the actual verification.

`Method Declaration`		
$\mathcal{M}$	::=	*sig* { *baseblks recurblk* `return` }
sig	::=	*name*(`iter`) | *name*(`iter`, $x_1, \ldots, x_n$)
`Base Case`		
baseblks	::=	*baseblk* | *baseblks baseblk*
baseblk	::=	`if` (*bcond*$_j$) { *bblock*$_j$ `return`}
bblock$_j$	::=	*block*
`Recursive Case`		
recurblk	::=	*rblock calls*
rblock	::=	*block*
calls	::=	*call* | *calls call*
call	::=	`if` (*rcond*$_j$) $\mathcal{M}$(`iter`$-> l_j$)
`Statements and Expressions`		
block	::=	*stmt* | *block stmt*
stmt	::=	*ifthen* | *du* | `skip`
bcond$_j$	::=	*expr*
rcond$_j$	::=	*expr*

Table 4.1: Syntax for Tail-Recursive methods

4.2 Tail-Recursive Methods

Recall from Section 2.3, the window-based abstraction for data structures. In what follows, we use $w(v_{\mathtt{I}})$ to denote the window with $v_{\mathtt{I}}$ as its root node. The syntax of tail-recursive methods is formally stated in Table 4.1. As before, we use italicized text for non-terminals and text in typewriter font for terminals in the grammar. As in Section 3.3, we define $\mathcal{M}$ to have a signature *sig*, and a body. The method body is sub-divided into base case blocks (*baseblks*), and a recursive block. The semantics are as follows: $\mathcal{M}$ first evaluates *bcond*$_j$ over the nodes in $w(v_{\mathtt{I}})$, and if *bcond*$_j$ is *true*, $\mathcal{M}$ performs the actions within *bblock*$_j$ followed by a `return`. If none of the *bcond*$_j$ evaluate to *true*, then $\mathcal{M}$ performs the actions within *recurblk*. This involves destruc-

tive updates ($rblock$) to the nodes within $w(v_{\mathtt{I}})$ followed by recursive visits to successors of $v_{\mathtt{I}}$ for which $rcond_j$ evaluates to $true$ (over the possibly updated $w(v_{\mathtt{I}})$). Block statements ($block$) are recursively defined to consist of conditional statements ($ifthen$), destructive update statements (du), or empty statements (skip). The syntax for the non-terminals $expr$, du, and $ifthen$ is the same as that for iterative methods (See Table 3.1). We assume that any calls to methods other than $\mathcal{M}$ are inlined within the body of $\mathcal{M}$. In contrast to standard programming languages, we specifically disallow loops, and advanced features like pointer arithmetic and arbitrary address manipulation.

Of special note is the syntactic restriction that allows $\mathcal{M}$ to recursively visit each successor of $v_{\mathtt{I}}$ at most once. This is both syntactically enforced, and statically checked by analyzing $\mathcal{M}$. Thus, a case where $\mathcal{M}$ visits the l_1-successor of $v_{\mathtt{I}}$, followed by updating $v_{\mathtt{I}}$ so that $v_{\mathtt{I}}.l_2 = v_{\mathtt{I}}.l_1$, followed by visiting $v_{\mathtt{I}}.l_2$ (which is now the same as $v_{\mathtt{I}}.l_1$) is detected at compile-time and disallowed. Since $\mathcal{M}$ is tail-recursive, $\mathcal{M}$ effectively visits any $v_{\mathtt{I}}$ at most once (since no work is done when $\mathcal{M}$ returns). Thus, $\mathcal{M}$ trivially satisfies the bounded updates property, as each $w(v_{\mathtt{I}})$ is destructively updated at most once.

Algorithm 4.1 shows the procedure for compiling $\mathcal{M}$ that respects the syntax specified in Table 4.1 into $\mathcal{A}_{\mathcal{M}}$. $\mathcal{A}_{\mathcal{M}}$ runs on a composite tree $\widetilde{T}_c$; thus, each $\sigma \in \Sigma$ is a pair of windows (w_i, w_o). We initialize the states Q, the symbols Σ, the initial state q_0, and the accepting states Φ in Line 3. The states $acc, rej, init$ have their usual meanings as an *accept* state, a *reject* state, and an *initial* state respectively.

The states in Q_Σ are used to check if $w(v_\mathtt{I})$ is consistent with the window(s) read at predecessors of $v_\mathtt{I}$. If $|w(v_\mathtt{I})| = 1$, then Q_Σ is empty, and then Lines 3-7 are skipped. Otherwise, we reject those state/symbol pairs that correspond to an inconsistent annotation at neighboring nodes in $\widetilde{T}_c$. If a state/symbol pair is consistent, we add it to the map H in Line 6.

Each state in Q_Σ is a pair of sub-windows (r_i, r_o). If $\mathcal{A}_\mathcal{M}$ reads the symbol $w_c = (w_i, w_o)$, then it remembers $r_i = succ(j, w_i)$ and $r_o = succ(j, w_o)$ in its next state along the j^{th} successor of w_c. If the height of a window w_i is h, let $head(w_i)$ denote the sub-window of height $h - 1$ rooted at the root of w_i. Thus, once $\mathcal{A}_\mathcal{M}$ reads a new symbol (x_i, x_o) as a successor of the symbol w_c, it checks if $x_c = (x_i, x_o)$ is consistent with w_c by checking if $head(x_i)$ is the same as r_i.

Thus, upon reading a node (x_i, x_o) in the state (r_i, r_o), we define the function consistent as *true* iff $head(x_i) = r_i$.

Note that the action of $\mathcal{M}$ at w_i (in w_c) changes the values of the nodes that are also a part of x_c. Thus, when $\mathcal{A}_\mathcal{M}$ arrives at x_c, some of its nodes are already modified. However, this information is encoded in the r_o component that $\mathcal{A}_\mathcal{M}$ remembers in its state. To compute the current and correct value of the input window at node x_c, we splice r_o with $tail(x_o)$ (or the leaf nodes of x_o). We denote this splicing operation by $r_o \hookrightarrow x_o$.

We explore all symbols and identify those that correspond to $\mathcal{M}$ entering any of the base cases. For the $bblock_j$ block of statements within the j^{th} base

Algorithm 4.1: CompileTailRecursive

```
1  begin
2      W(z) = all windows of height z
3      Σ:=W × W, q0:=init, Q:={init, acc, rej} ∪ QΣ, Φ:={acc}, H :={}
       /* Note that σ = (wi, wo) */
4      foreach σ in Σ, q in QΣ do
           /* Reject inconsistent (q, σ) */
5          H :=H ∪ {(σ, init)}
6          if (consistent(σ,q)) then H :=H ∪ (σ, q)
7          else reject(q,σ)
8      RS:=Σ
9      foreach baseblk in baseblks, σ in Σ do
10         if (σ ⊨ bcondj) then
11             RS :=RS \ {σ}
12             f_bblockj := computeBlock(bblockj)
               /* σ mimics bblockj? */
13             if (faithfulUpdate(wi, wo, q, bblockj)) then
14                 foreach q in H(σ) do accept(q,σ)
15             else
16                 foreach q in H(σ) do reject(q,σ)
17     foreach σ in RS do
18         f_rblock :=computeBlock(rblock)
19         if (faithfulUpdate(wi, wo, q, rblockj)) then
               /* σ mimics rblock */
20             foreach q in H(σ) do
21                 foreach j in {1, ..., K} do
22                     if (σ ⊨ rcondj) then
23                         next[j]:= computeState(σ)
24                     else
25                         next[j]:=acc
26                 δ:= δ ∪ {(q, σ, next)}
27         else
               /* σ does not mimic rblock */
28             foreach q in H(σ) do reject(q,σ)
29 end
```

case, we can compute a function f_{bblock_j} that represents the composition of the functions for individual statements within $bblock_j$. If the update encoded by σ is faithful to f_{bblock_j}, we accept all consistent states for this symbol (Line 14), else we reject them (Line 16).

The function $\mathsf{faithfulUpdate}(w_i, w_o, q, block)$ for a state $q = (r_i, r_o)$ is defined to be $true$ if $f_{block}(r_o \hookrightarrow w_i) = w_o$, and $false$ otherwise. Thus, for the base case this is equivalent to checking if $f_{bblock_j}(r_o \hookrightarrow w_i) = w_o$.

Once all *baseblk* statements are processed, the remaining symbols (in the set RS) correspond to symbols for which $\mathcal{M}$ enters the recursive case. For each symbol $\sigma \in RS$, we check if the update encoded by σ is faithful to f_{rblock}. If not, we reject all consistent states for that σ (Line 28). If yes, we identify the successors of $v_{\mathtt{I}}$ within $w_o(v_{\mathtt{I}})$ that $\mathcal{M}$ would visit (by virtue of $rcond_j$ evaluating to $true$), and transition to the appropriate state (from Q_Σ) for those successors (Line 23). The remaining successors are not visited by $\mathcal{M}$, and hence, we simply transition to acc for these (Line 25). We use `reject`(q,σ) as a macro to indicate adding a transition of the form $(q, \sigma, (rej, \ldots, rej))$ to δ (similarly `accept`). The function computeBlock composes the functions f_s for individual statements s within a block statement. computeState returns the state in Q_Σ that encodes the value of the current pair of windows (σ). Essentially, for ease of implementation, we map each (r_i, r_o) pair to a unique string, which is returned by computeState. As a final step (not shown in the algorithm), we add self-loops to the acc and rej states, making them "trap" states.

Tail-recursive methods can be simulated by iterative methods that use loops. Thus, in theory, we could translate a tail-recursive method specified here into an iterative method, and verify it using techniques in Chapter 3. However, in Chapter 3, we assume that there exists an oracle that predicts if a given $\mathcal{M}$ satisfies the bounded updates property. In contrast, the syntax presented in Table 4.1 *ensures* that methods written with this syntax satisfy the bounded updates property. Thus, we eliminate most of the manual effort that was previously required. Furthermore, we are also able to handle a larger class of recursive methods beyond just tail-recursive methods, as seen in the next section.

4.3 Decidable Syntactic Class of Recursive Methods

In the previous section, we identified a syntactic class containing tail-recursive methods, for which automatic verification was possible as the corresponding method automata obtained were finite state tree automata. In this section, we show that finite state method automata can also be obtained for more general recursive methods, subject to nearly identical syntactic constraints as in Table 4.1. In contrast to tail-recursive methods, non-tail recursive methods can re-visit a node between recursive calls to its successors, and perform destructive updates. We make the same assumptions as for tail-recursive methods that: a) methods do not use global pointers, b) methods use a single pointer argument during recursive invocation, and c) at any node, for any given successor s, $\mathcal{M}$ is invoked with s as an argument at most once. Consider

a tree where the maximum out-degree of any node is K. A recursive method $\mathcal{M}$ satisfying the above assumptions visits a node with K out-going edges: a) the first time when called from the parent node of n, and b) K times after each recursive call returns. Thus, if $\mathcal{M}$ makes a constant number of destructive updates per visit, then the total number of destructive updates to any node is $O(K+1)$. However, for a given tree, K is a constant, and thus the total number of destructive updates is still bounded by a constant. Thus, such a $\mathcal{M}$ satisfies the bounded updates property.

The syntax for general recursive methods is largely similar to the syntax presented in Table 4.1. Instead of a single *recurblk*, the there are up to K *recurblk* statements, where each *recurblk* statement consists of a block of statements ($rblock_j$) followed by a recursive call to the j^{th} successor of $v_{\mathtt{I}}$. We show the differing parts in Table 4.2. All the other definitions remain the same.

The overall scheme for compilation into $\mathcal{A}_{\mathcal{M}}$ for the class in Table 4.2 is similar to the one used for compiling tail-recursive methods. In Algorithm 4.1, all destructive updates by a tail-recursive method on a given window can be composed into a single destructive update described by the function f_{block} using computeBlock. This is not possible for a method belonging to Table 4.2, since it performs $K+1$ distinct blocks of updates. However, by altering the way we define the composite tree, we can use an algorithm very similar to Algorithm 4.1 to compile $\mathcal{M}$ into $\mathcal{A}_{\mathcal{M}}$. We first observe that if we record the actions of such a $\mathcal{M}$ during a depth-first traversal at each node in the underlying

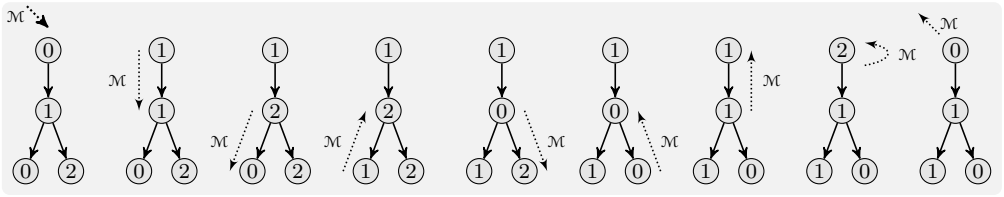
`Method Declaration`		
$\mathcal{M}$	::=	sig { $baseblks$ $recurblks$ `return` }
sig	::=	$name$(`iter`) | $name$(`iter`, $x_1, \ldots, x_n$)
`Base Case`		
$baseblks$	::=	$baseblk$ | $baseblks$ $baseblk$
$baseblk$	::=	`if` ($bcond_j$) { $bblock_j$ `return`}
$bblock_j$	::=	$block$
`Recursive Case`		
$recurblks$	::=	$recurblks$ $recurblk$ | $recurblk$
$recurblk_j$	::=	$rblock_j$ $call_j$
$call_j$	::=	`if` ($rcond_j$) $\mathcal{M}$(`iter`$- > l_j$)
`Statements and Expressions`		
$block$	::=	$stmt$ | $block$ $stmt$
$stmt$	::=	$ifthen$ | du | `skip`
$bcond_j$	::=	$expr$
$rcond_j$	::=	$expr$

Table 4.2: Decidable Syntactic Class for Recursive Methods

data structure D, we obtain an annotated D', where every node of D' is a $K+1$ tuple of values. We illustrate this with an example in Example 4.3.1.

Example 4.3.1. Consider the method $\mathcal{M}$ = changeData shown in Figure 4.3b. $\mathcal{M}$ changes the data value of each node in the input tree. We assume that the data domain $\mathcal{D}$ is the set $\{0, 1, 2\}$, and use incrMod3 as a macro to replace three conditional assignments that specify *modulo*-3 increment. Figure 4.3a shows the actions of $\mathcal{M}$ on an input tree, while Figure 4.3c shows the input tree annotated with the actions of $\mathcal{M}$.

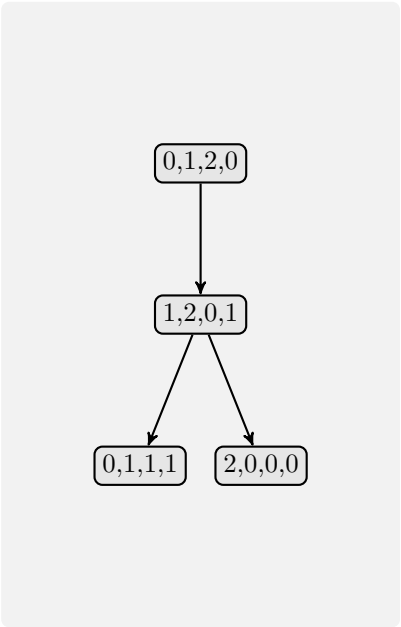
We use the intuitive idea that the action of a depth-first recursive method $\mathcal{M}$ generates an annotation similar to that in Example 4.3.1 on the underlying tree. We define the composite tree $\widetilde{T}_c$ s.t., each node w_c in $\widetilde{T}_c$



(a) Actions of $\mathcal{M}$ on tree $\widetilde{T}_i$

```
changeData (iter) {
   /* base case */
   if ((iter->l1 == null) &&
       (iter->l2 == null)) {
      incrMod3 (iter− >data);
      return;
   }
   /* recursive case */
   incrMod3 (iter− >data);
   if (iter− >l1 ≠ null) {
      changeData(iter− >l1);
   }
   incrMod3 (iter− >data);
   if (iter− >l2 ≠ null) {
      changeData(iter− >21);
   }
   incrMod3 (iter− >data);
   return;
}
```

(b) Method changeData



(c) Annotated Composite tree $\widetilde{T}_c$

Figure 4.3: Composite Tree Encoding Action of a Recursive Method

is a $(K+2)$-tuple of the form $(w_0, w_1, \ldots, w_K, w_{K+1})$. Here, $w_i = w_0$, and $w_o = w_{K+1}$.

Algorithm 4.2 shows the algorithm to compile a method with non-tail recursion. Note that it is largely similar to Algorithm 4.1, except for Lines 17-31. Note that the definition for faithfulUpdate slightly changes. If $j = 0$, faithfulUpdate is *true* iff $f_{rblock_j}(r_o \hookrightarrow w_0) = w_1$ (since w_0 is the same as w_i). For $j > 0$, faithfulUpdate is *true* iff $f_{rblock_j}(w_{j-1}) = w_j$. We use the computeBlock procedure to compute f_{rblock_j} corresponding to each $rblock_j$ statement. If any of the faithfulUpdate checks fails, we reject the entire symbol w_c (Line 22). If all pairs of windows encode faithful updates, then we proceed as before.

We formalize the correctness of Algorithm 4.2 in Theorem 4.3.1. We can informally prove the theorem using a similar argument as in Section 3.4.2.

Theorem 4.3.1 (Correctness)**.** *$\mathcal{A}_{\mathcal{M}}$ derived using Algorithm 4.2 has the following properties: (a) $\mathcal{A}_{\mathcal{M}}$ is a finite-state tree automaton,* i.e.*, a finite Q and Σ, (b) $\mathcal{A}_{\mathcal{M}}$ accepts $\widetilde{T}_c = \widetilde{T}_i \circ \widetilde{T}_o$ iff $T_o = \mathcal{M}(T_i)$,* i.e. *$\mathcal{A}_{\mathcal{M}} \bowtie \mathcal{M}$, and (c) $\mathcal{A}_{\mathcal{M}}$ rejects $\widetilde{T}_c$ if $\widetilde{T}_i$ is inconsistent.*

Finally, we remark that we can extend our basic approach to handle returning data values and pointers that are not modified by the parent node. Also, we can allow methods to have a limited access to predecessor nodes up to a bounded distance, and allowing more than one pointer argument in the method signature, with the restriction that all arguments are contained

Algorithm 4.2: CompileGeneralRecursive

```
1  begin
2      W(z) = all windows of height z
3      Σ:=W^{K+2}, q_0:=init, Q:={init, acc, rej} ∪ Q_Σ, Φ:={acc}, H :={}
       /* Note that σ = (w_i, w_o) */
4      foreach σ in Σ, q in Q_Σ do
           /* Reject inconsistent (q, σ) */
5          H :=H ∪ {(σ, init)}
6          if (consistent(σ,q)) then H :=H ∪ (σ, q)
7          else reject(q,σ)
8      RS:=Σ
9      foreach baseblk in baseblks, σ in Σ do
10         if (σ ⊨ bcond_j) then
11             RS :=RS \ {σ}
12             f_{bblock_j} := computeBlock(bblock_j)
               /* σ mimics bblock_j? */
13             if (faithfulUpdate(w_i, w_o, q, bblock_j)) then
14                 foreach q in H(σ) do accept(q,σ)
15             else
16                 foreach q in H(σ) do reject(q,σ)
17     foreach σ in RS do
18         rejected:=false
           /* Check if σ is faithful to each rblock_j */
19         foreach j in {1, ..., K} do
20             f_{rblock_j}:= computeBlock(rblock_j)
21             if (¬ faithfulUpdate(w_{j-1}, w_j, q, rblock_j)) then
22                 foreach q in H(σ) do reject(q, σ)
23                 rejected := true
24                 break
25         if (¬rejected) then   foreach q in H(σ) do
26             foreach j in {1, ..., K} do
27                 if (σ ⊨ rcond_j) then
28                     next[j]:= computeState(σ)
29                 else
30                     next[j]:=acc
31             δ:= δ ∪ {(q, σ, next)}
32 end
```

within a window. Each of these extensions still preserves the bounded updates property. The algorithms for translation can be modified accordingly to accommodate these extensions.

4.4 Complexity Analysis

The complexity of our technique is proportional to the complexity of checking emptiness of the product automaton $\mathcal{A}_p$. For acceptance conditions such as the Büchi condition, and parity condition with a small finite number of colors, this is polynomial in the number of states of $\mathcal{A}_p$. This is in itself linear in the size of $\mathcal{A}_{\mathcal{M}}$ (denoted $|\mathcal{A}_{\mathcal{M}}|$), $|\mathcal{A}_{\varphi}|$ and $|\mathcal{A}_{\neg\psi}|$.

The number of states in $|\mathcal{A}_{\mathcal{M}}|$ is linear in the size of $\mathcal{M}$ and $|\Sigma|$. Recall that all automata have the same input alphabet Σ, and all automata sizes are dominated by $|\Sigma|$. For tail-recursive methods, Σ is the set of all pairs of windows. Suppose the size of the window is h (number of nodes in the window). For a data domain of size $|\mathcal{D}|$, the number of distinct nodes, the pointer fields for which range over $\{0, \ldots, h-1, *, \emptyset\}$ is $n = |\mathcal{D}|.(h+2)^K$. The number of possible windows is thus n^h. Since h is fixed, $|\Sigma|$ is thus $O(|D|^h.h^{K.h})$, *i.e.*, polynomial in $|\mathcal{D}|$, and exponential in the window size h and the degree K.

Remark 1: We note that for purely structural properties, $\mathcal{D}$ can be often abstracted to a single symbol, and in practice K is small. Also, a large h in practice is uncommon, as most methods are written so that their action is local in nature.

Remark 2: We note that for both input and output windows, a large number of windows are illegal, *i.e.*, for example, in the case of input windows, disconnected windows are not possible, and in the case of output windows (if the desired output is a tree), windows that violate acyclicity, treeness, are not valid. Thus a large number of windows can be pruned away reducing the size of Σ.

Remark 3: It is possible to further reduce the size of Σ by introducing predicates that divide windows into equivalence classes. We term this *symbol clustering*. The product and emptiness operations can be re-defined and interpreted over symbol clusters instead of actual symbols. This would give orders of magnitude reduction in the size of Σ, and hence the sizes of the automata.

In summary, the overall complexity of our technique is polynomial in the size of the method, the sizes of the specifications, the data domain and exponential in the window size and the degree of the data structure. In practice, since window size and degree are small positive integers, we can safely say that our technique has *polynomial* complexity in $|\mathcal{M}|$, $|\mathcal{A}_\varphi|$ and $|\mathcal{A}_{\neg\psi}|$.

In Chapter 5, we demonstrate the capability of our technique with the help of a prototype tool that is able to verify real-world methods for an interesting set of specifications.

4.5 Bibliographic Notes

Most of the material presented in this chapter appears in [41], and some of the required background can be found in [42]. Most of the comparison with related work can be found in Section 3.7; we discuss the work relevant to verifying recursive methods here.

Recursion is usually one of the big challenges in automatic software verification. Modeling recursive methods typically requires reasoning about call stacks. In fact, many popular and highly useful software verification techniques choose to not handle recursion. For instance, `Java PathFinder`, which is a highly successful software model checking tool does not handle recursion [80]. The authors report that recursion modeled as a process concept in the PROMELA language (which is an input language for the SPIN model checker), could be done in theory, but initial experiments showed that it made for inefficient verification.

Analysis of recursive methods based on model checking, is usually framed as the model checking problem for pushdown systems [64, 18, 134]. Many of these techniques leverage the fact that the state space of a pushdown system is a regular set [26]. However, all of these techniques assume that the system is finite state except for the additional pushdown structure. In case of possibly unbounded linked data structures, the system state includes the state of the data structure, and hence the system is *not* a finite state system as required in these approaches. Interprocedural dataflow analyses such as those explored in [63] track flow facts across method calls. Typically, such dataflow

facts are elements of a finite lattice, whereas such a construction would fail for potentially infinite linked data structures.

The most pertinent work for verifying recursive methods on data structures is that of interprocedural shape analysis [126] of recursive methods that manipulate lists. This summarization-based algorithm iteratively annotates each program point with a set of 3-valued logical structures in a conservative fashion. These 3-valued structures are bounded size over-approximations of the possibly unbounded lists. To model recursion, the analysis summarizes activation records in the same way that it summarizes linked list elements. This approach improves the precision with respect to previous approaches such as [130]. The approach is parameterized by a predefined set of library properties; though this eases the burden of specification, it can lead to poor results on programs that require other properties to be tracked.

[127] extends the authors' previous work to cutpoint-free programs - programs in which reasoning on a procedure call only requires consideration of context reachable from the procedure parameters. Here, the analysis computes procedure summaries as transformers from inputs to outputs while ignoring parts of the heap not relevant to the procedure. [90] presents an approach for context-sensitive interprocedural shape analysis that uses relational representation for the approximation of the evolution of linked data structures. The approach is able to handle lists and binary trees in a conservative fashion. [72] presents an analysis that makes use of spatial locality and is able to handle cyclic and shared data structures.

The key difference between shape-analysis based approaches and our technique is exactness and expressivity. Shape analysis can handle arbitrary programs, properties (albeit restricted to those provided as combinations of core predicates and instrumentation predicates), and is this highly expressive. However, it uses of 3-valued logic over first-order structures, and is thus conservative, and thus, prone to inaccurate results due to false positives. Moreover, the theoretical complexity of the decision procedure is doubly exponential, and thus its scalability is subject to engineering optimizations. On the other hand, we only model methods that satisfy a certain syntax, and properties that can be expressed as tree automata. While the latter is typically quite expressive, the former restriction is a limitation. It may be possible to combine the abstraction offered by the interprocedural shape analysis based approaches with our exact verification technique, to obtain a larger class of methods that can be verified more efficiently and precisely.

In [4], the authors describe *visibly pushdown tree languages* as a subclass of context-free tree languages for explicitly modeling calls/returns in a recursive program. The authors define *visibly pushdown automata* (*VPTA*) that operate on trees representing the branching behavior of structured programs. Non-deterministic *VPTA* have desirable closure properties and an *EXPTIME*-complete decision procedure. Though this work focuses on a different problem, application of such automata for verification of methods operating data structures is worth investigation.

Chapter 5

Experimental Evaluation

In Chapters 3 and 4, we presented a solution framework to respectively verify the correctness of iterative and recursive methods on heap-allocated linked data structures. We have developed a prototype tool PRAVDA (Tool for **P**a**RA**meterized **V**erification of **DA**ta structures). We briefly explain the tool architecture followed by the results obtained using the tool.

5.1 Tool architecture

PRAVDA, written in `Java`, consists of two high-level modules. The first module is a parser-lexer scheme written in `Java-CUP`[87] and `jflex`[95], which parses the given method text and the given properties and builds a method automaton and pre/post-condition automata respectively. The second module is a library for manipulating tree automata, containing implementations of standard algorithms for product construction checking nonemptiness. This module also outputs a counter-example to the correctness of the method if the product automaton is nonempty.

One of the features of PRAVDA is its ability to detect important bugs such as: *null pointer dereferences*, *memory leaks*, and *double deletion* of nodes.

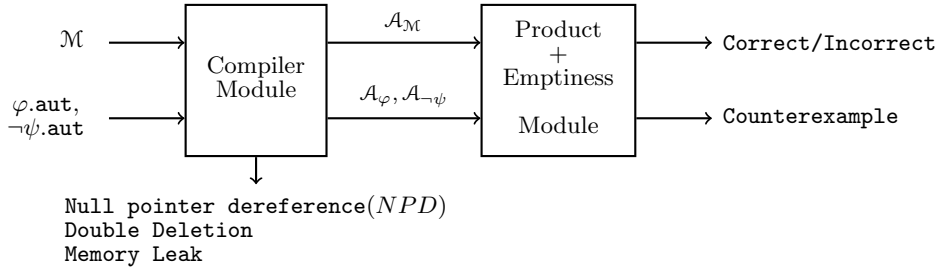


Figure 5.1: Architecture of PRAVDA

To ease the burden of specification from the user, these checks are built into the tool, and do not have to be explicitly specified. Furthermore, these checks are reported during the construction of the method automaton from the method description. The tool architecture is shown in Figure 5.1.

The current implementation of the tool is only a prototype, and lacks important engineering optimizations that can dramatically increase scalability by reducing the memory consumption. For instance, pruning the size of the method and automata to include only reachable states is a simple optimization that can lead to significant savings. We remark that further space savings are possible by using a symbolic representation for the transition tables of the various automata by leveraging the power of *BDD*s or *SAT* encodings [107, 28].

5.2 Experimental Results for Iterative Methods

Table 5.1 displays some of the notable results obtained using PRAVDA. The method `AddSelfLoop` is a very small method that introduces a self-loop on the first node of a linked list. `AddSelfLoopTail` introduces a self-loop on

Table 5.1: PRAVDA: Performance Results for Iterative Methods

Method	Memory Consumed (MB)	Time Taken (secs)	Specification	Result
On Lists:				
`AddSelfLoop`	1.3	0.4	Acyclicity	`Incorrect`
`AddSelfLoopTail`	1.2	0.4	Acyclicity	`Incorrect` `NPD`
`InsertAtTail`	10	1.2	Acyclicity,	`Correct`
`SubstituteAll(a,b)`	1.5	0.4	Acyclicity,	`Correct`
			$\mathbf{AG}(\texttt{iter}- > \texttt{data} \neq a)$	`Correct`
On Trees:				
`GetListofRightChildren`	12	1.2	Acyclicity	`Correct`
`DeleteLeaf`	530	15	Acyclicity,	`Correct`
			$\mathbf{EF}(\texttt{iter} \neq 0)$	`Correct`
`SubstituteAll(a,b)`	250	8	Acyclicity,	`Correct`
			$\mathbf{AG}(\texttt{iter}- > \texttt{data} \neq a)$	`Correct`
`InsertNode`	2.5K	221	Acyclicity	`Correct`

the tail node of a linked list. Method `InsertAtTail` inserts a node at the tail of the linked list. Method `SubstituteAll(a,b)` changes the data values of all nodes that have data value `a` to the data value `b` in both trees and lists. Method `GetListOfRightChildren` returns a list-like representation of the rightmost path in a binary tree by setting the 0^{th} child pointer of each (rightmost) node to `null`. The `DeleteLeaf` method deletes a leaf node if it is not the same as the root node. The `InsertNode` method inserts a node at some location in a binary tree. We primarily checked for whether acyclicity was preserved by the methods.

In some cases, post-conditions for methods involved reasoning over data

values of nodes. In these cases, the pre-condition was assumed to be *true*. For instance we checked for the property "there does not exist a node with data value `a`" ($\mathbf{AG}(\texttt{iter}->\texttt{data} \neq a)$) and "there exists at least one node in the data structure" ($\mathbf{EF}(\texttt{iter} \neq 0)$).

All experiments were performed on an AMD Athlon 64X2 4200^+ system with 6GB RAM. The memory consumption is proportional to the sizes of the automata constructed by our technique. There is a direct correlation between the memory consumption and (a) the size of the window required to verify (or falsify) a method, and (b) the branching degree of the underlying data structure. The window size for all methods except `InsertNode` was 2, while that for `InsertNode` is 3. We can observe that as the window size and branching degree increase, the memory consumption increases exponentially. The time taken, on the other hand, increases linearly with the size of the automata. This is expected as the complexity of checking nonemptiness is polynomial (quadratic in most cases).

5.3 Experimental Results for Recursive Methods

We show some of the notable results in Table 5.2. We can see in Table 5.2 that the time required to construct $\mathcal{A}_{\mathcal{M}}$ is a fraction of the total time taken, but the memory consumption is a sore spot, as we use explicit representation for Σ. As before, we can observe that the time taken and memory consumption dramatically increases with increasing degree K. Also, with increase in the size of the window, the time taken and memory consumed increases,

as can be seen for the method `InsertNode` that uses a window of size 3, in contrast to other methods that use a window of size 2.

Table 5.2: PRAVDA: Performance Results for Recursive Methods

Method	Specification	Time (secs) $\mathcal{A}_{\mathcal{M}}$	Time (secs) Total	Mem. (GB)
On Linked Lists:				
`DeleteNode`	Acyclicity	0.3	1.3	0.02
`InsertAtTail`	Acyclicity	0.01	0.8	0.001
`InsertNode`	Acyclicity	0.4	1.6	0.04
On Binary Trees:				
`InsertNode`	Acyclicity	15	329	2.5
`SubstituteAll(a,b)`	Acyclicity	5	26	0.3
	$\mathbf{AG}(\texttt{iter}- > \texttt{data} \neq a)$	5	27	0.4
`DeleteLeaf`	Acyclicity	12	48	0.6

The methods that we present have similar functionality as the iterative methods in Section 5.2, with the difference that these employ recursion. As before, they are commonly found methods in implementations of linked list and tree libraries written in `C`, adapted to our syntax.

5.4 Counterexample Generation

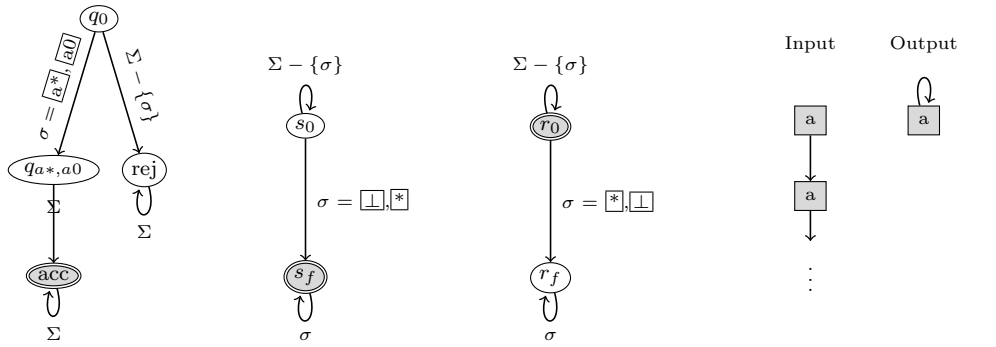
If $\mathcal{A}_p$ is found to be nonempty, the tree $\widetilde{T}_c$ witnessing its nonemptiness is extracted from the transition diagram δ_p of $\mathcal{A}_p$ using standard techniques. By projecting $\widetilde{T}_c$ onto its first and last components, we can obtain trees $\widetilde{T}_i$ and $\widetilde{T}_o$ respectively, and from these obtain a candidate input tree T_i and an output tree T_o. T_i represents an input to $\mathcal{M}$ for which the "bad" output T_o is

generated, *i.e.*, a concrete counterexample to the correctness of $\mathcal{M}$.

Figure 5.2: Method `AddSelfLoop`

```
1: DataDomain :={ a }
2: AddSelfLoop (iter) {
3:    iter->data :=a;
4:    iter->l_0 :=iter;
5: }
```

Example 5.4.1. Consider the method shown in Figure 5.2. This method introduces a self-loop on the first node of the linked list that it operates on. The product of the method automaton (Figure 5.3a), the pre-condition automaton (Figure 5.3b) and the negated post-condition automaton (Figure 5.3c) is nonempty, and accepts the super-imposition of the input/output graphs shown in Figure 5.3d.



(a) $\mathcal{A}_{\mathcal{M}}$ **(b)** $\mathcal{A}_{\varphi}$ **(c)** $\mathcal{A}_{\neg\psi}$ **(d)** Counter-example

Figure 5.3: Counter-example from Product Automaton

Part III

Analysis of Concurrent Libraries

Outline. Concurrent software presents some of the greatest challenges to software verification. Thread safety violations such as data races, atomicity violations, deadlocks, and starvation compromise the reliability and stability of multi-threaded software. In this part of the dissertation, we focus on static techniques for deadlock analysis in concurrent software libraries. Such libraries, being open systems, require techniques that exceed the scope of deadlock analysis techniques for conventional software that is typically treated as a closed system.

In what follows, we first provide the necessary background on concurrent programming, static analysis-based techniques and synchronization primitives in Chapter 6. We discuss *deadlockability analysis* - a deadlock analysis technique for libraries in Chapter 7, with a focus on lock-based synchronization. In Chapter 8, we extend deadlockability analysis to reason about deadlocks in libraries that use signaling-based synchronization with `wait-notify` constructs. Finally, we present experimental results in Chapter 9.

Chapter 6

Background

Design and verification of concurrent software is one of the grand challenges in computer science. Concurrency verification poses a number of obstacles that sets if apart from any other formal verification problem. It is a well-researched area with a vast and ever-expanding literature. For a comprehensive treatment of some of the earlier techniques in concurrency verification, please see [38].

Most concurrent software is built assuming an interleaved model of execution. This model allows for nondeterministic scheduling of individual threads of computation. Hence, exhaustive reasoning about correctness typically requires building the entire global state graph, which is astronomical in size due to an exponential number of interleavings. While techniques based on model checking coupled with partial-order reduction [119, 67] and symmetry reductions [59] or combinations of these techniques [53] have ameliorated this problem to a certain degree, analyzing real-life software, which is a source of many bugs, is a computationally formidable task.

Verification techniques based on static analysis have emerged as a viable alternative to model checking. Model checking uses state-based opera-

tional semantics of programs, in converting a program to an equivalent Kripke structure. In contrast, static analysis techniques typically rely on an abstract interpretation of the program. Thus, depending on the kind of analysis to be performed, the effect of each program statement is *interpreted* accordingly. This is usually achieved by interpreting the state of the program as a dataflow fact, and re-defining the program statements as operators that modify the dataflow facts.

Model checking, which searches for the existence of certain temporal patterns with the state-space of a program, can also be classified as a static technique, as it does not involve executing the program. However, the term static analysis is conventionally associated with techniques that infer information from a static representation of the program, such as the program text, or some intermediate representation, rather than from the finite or finitized state-space of the program.

The biggest advantage for analyzing concurrent software statically lies in the ability to perform thread-modular analysis. In such an analysis, the dataflow facts for the statements in a thread are combined to obtain a thread summary. Thread summaries are then meaningfully combined in order to reason about concurrent executions of such threads. Thus, such an analysis reduces complexity on two distinct indices: (1) by abstract interpretation of the statements in a thread it reduces the complexity of local reasoning in a thread, and (2) by safe approximation of possible schedules of thread execution, it reduces the complexity introduced by interleaving.

However, the biggest price paid by static analysis is in its lack of precision. Most static analysis tools such as data race detection tools are plagued by too many *false positives*. A false positive is a suspected error in the abstract interpretation of the program, which is infeasible in the concrete program. In other words, a false positive is an error introduced by the abstraction, and not a concrete error in the program. It is possible to improve the precision of static analysis tools; however, any such effort has to be aware of the fine balance between precision and scalability.

In presence of recursion, it has been shown that dataflow analysis for computing simultaneous reachability of locations in a pair of threads for even the simplest of programs is undecidable [124]. In other words, there is an upper limit on the available precision for concurrent data flow analysis. Nevertheless, due to the remarkable scalability vis-á-vis model checking based approaches, static analysis based tools are widely gaining currency in software verification practice.

In this part of the dissertation, we focus on the application of static analysis to modular verification of concurrent software libraries. In particular, we focus on the problem of deadlock prediction and detection. Analyzing concurrent libraries has its own set of challenges in addition to the challenges faced by concurrency verification. Modular software development principles promote well-encapsulated concurrent software libraries that can then be used by multiple programs (referred to as clients). Such libraries may be informally specified by the application programming interface (API) documents

that accompany their object (or source) code. However, information hiding obscures internal synchronization details of the libraries from the client developers. This is further worsened by the fact that such details of synchronization are rarely made a part of the API. This results in developers invoking library methods in potentially hazardous fashion, exposing the client code to thread safety violations such as *data races* and *deadlocks*. In what follows, we give a brief overview of static analysis, a programming language model, and introduce *deadlockability analysis* as a way to analyze deadlocks in concurrent libraries.

6.1 Static Analysis

Static code analysis techniques have emerged as scalable alternatives within the purview of formal approaches to exhaustive software verification. Model checking, which searches for the existence of certain temporal patterns with the state-space of a program, can also be classified as a static technique, as it does not involve executing the program. However, the term static analysis is conventionally associated with techniques that infer information from a static representation of the program, such as the program text, or some intermediate representation, rather than from the finite or finitized state-space of the program.

We now give a brief introduction to dataflow analysis, which is a type of a static analysis, and provide a quick glossary of terms that we will use throughout the rest of this dissertation. Most of this material is adapted from

[114].

Control-flow Graph. A control flow graph (*cfg*) is a directed graph (V, E) where the V is a set of control-flow graph nodes, and each edge in E is labeled with a program statement. Each control-flow graph has a designated *entry* vertex *entry*, and a designated *exit* vertex *exit*.

Partial Order, Lattices. A partial order is a structure $(X, \sqsubseteq)$, where X is a finite set, and $\sqsubseteq$ is a reflexive, transitive and anti-symmetric binary relation on X. If $S \subseteq X$, then $y \in X$ is an *upper bound* for S if $\forall s \in S, s \sqsubseteq y$. An upper bound $\hat{y}$ is a *least upper bound* (also called *join*) for S if for every y that is the upper bound for S, $\hat{y} \sqsubseteq y$. We can similarly define a *lower bound* and the *greatest lower bound* (also called *meet*). Given a subset $\{a, b\}$, we denote the *join* by $a \sqcup b$, and *meet* by $a \sqcap b$. A partially ordered set X (with the the partial order $(X, \sqsubseteq)$) forms a lattice if the meet and join exist and are contained in X for every pair of elements in X. A finite lattice has a unique least element or *infimum*, denoted as $\bot$ or *inf*, and a unique greatest element or *supremum*, denoted as $\top$ or *sup*.

Monotone Framework. A function $f : X \to X$ is *monotone* if $\forall x, y \in X : x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$. An element x is a fixpoint for f if $f(x) = x$. By the Knaster-Tarski theorem it follows that for every finite lattice, every monotone function has a unique least fixpoint computed as $\mathit{lfp}(f) = \sqcup_{i \geq 0} f^i(\bot)$ [36].

A set of *dataflow facts* $(L, \sqsubseteq)$ forms a lattice, and thus has a well-defined associated partial order $\sqsubseteq$, well-defined meet ($\sqcap$) and join ($\sqcup$) operators. The inputs to a dataflow analysis are the control-flow graph $cfg_P(V, E)$ of a program P. Let u, v be control-flow graph nodes, and let s be a program statement. The analysis is formulated as the following set of equations:

$$fact(v) = \begin{cases} i & \text{if } v \in I \\ \square\{f_s(fact(u)) | (u \xrightarrow{s} v) \in E'\} & \text{otherwise} \end{cases}$$

Here, $fact(u)$ represents a dataflow fact at the control-flow graph node u, $\square$ is either the join operator $\sqcup$ or the meet operator $\sqcap$, E' is either E or E^R (*i.e.*, the set of reverse edges in cfg_P), I is either $\top = fact(entry)$ or $\bot = fact(exit)$, and f_s is a transfer function associated with a program statement s.

If E' is E, we term the dataflow analysis as a *forward analysis*, and if E' is E^R, we term the dataflow analysis as a *backward analysis*. If $\square$ is $\sqcup$, we are interested in computing the *least sets* that solve the set of data-flow equations. These analyses are able to detect properties satisfied by *some* execution path within the program, and are also referred to as *may analyses*. On the other hand, if $\square$ is $\sqcap$, we are interested in computing the *greatest sets* that solve the equations. These analyses are able to detect properties satisfied by *all* execution paths within the program, and are also referred to as *must analyses*.

If the transfer functions f_s in the dataflow analysis are monotone, it follows from Knaster-Tarski theorem, that the least (or greatest) solutions to the set of dataflow equations exist, and can be computed iteratively, starting from $\perp$ (or $\top$). Historically, there have are two variant methods to solve dataflow equations: the MFP solution or the *Maximal Fixed Point* solution, and the MOP or the *Meet Over all Paths* solution[1].

Interprocedural Analysis. The monotone framework presented above only solves dataflow equations within a procedure, and hence is an instance of an *intraprocedural analysis*. Interprocedural analyses deal with analyzing flow information in the presence of functions (or methods, in `Java` parlance). In this work, we primarily make use of interprocedural analysis by means of computing function summaries [136]. Here, the key idea is to succinctly represent the behavior of a method parameterized by information about its arguments. The formalism introduced in [136] is limited to finite lattices of dataflow facts, which suits our purpose in this work.

There are two flavors for summary-computation based interprocedural analysis. In the *eager* approach, we try to compute an approximate *call graph*[2], and pre-compute method summaries in the reverse topological order of the

[1]For may analyses, we compute the Least Fixed Point, or analogously, do a Join Over all Paths, but historically analyses have focussed on must analyses, and hence it is common to call the techniques MFP or MOP solutions.

[2]A call graph is a directed graph representing the order in which methods are invoked in a program. The "top-level", *i.e.*, typically the `main` method is at the root of the graph, and for each method m, any method m' invoked in the body of m is a child of m in the call graph. Recursion leads to cycles in the call graph.

call graph. This ensures that intraprocedural analysis of a method m has the summaries of the methods that are invoked by m available at the point of invocation. On the other hand, the *lazy* or demand-driven approach computes function summaries as required.

Context-sensitivity. A context-insensitive analysis is one in which the information about calling states is combined at call sites, the procedure is analyzed only once using the combined information, and the resulting information about the set of return states is used at all return points. On the other hand, context-sensitive analysis uses the precise information (also known as context) at a call-site. In an interprocedural context-sensitive analysis, each call-site is treated differently, and the actual values of variables in the call-site are mapped to the formal parameters present in the method summary.

Flow-sensitivity. The dataflow analysis that we presented is flow-sensitive, *i.e.*, the flow information is influenced by the order in which statements appear in the program. In simple terms, in a flow-insensitive analysis the order in which statements appear in a program has no importance. Flow-insensitive analyses have application where a cheap and fast (albeit less precise) analysis is needed.

Lock-Graph Analysis. In Chapter 7, we outline a static lock-graph analysis that captures dependencies between lock-acquisitions in a program. In

terms of the concepts outlined above, our lock-graph analysis is a context-sensitive, flow-sensitive, interprocedural analysis. A dataflow fact in our analysis is a lock-order graph, which encodes the lock dependencies. For a given set of locks, the possible lock-order graphs forms a lattice under the partial order imposed by the subset ($\subseteq$) relation for the vertices and edges. The *sup* is the fully connected graph, while the inf is the graph where no two nodes are connected. The $join$ or $\sqcup$ operator computes the union of two graphs (union of the sets of vertices and edges respectively). We use the MFP method to compute least fixpoint solutions for the dataflow equations, and define transfer functions for statements that acquire or release locks. Finally, we note that a precise lock-graph analysis requires an *a priori* alias analysis. We start with a default context-insensitive, intraprocedural alias analysis offered by the static analysis framework that we use. However, as we wish to focus on precision, during the course of our lock-graph analysis, we track context-sensitive, interprocedural aliases for objects (lock variables) of interest.

6.2 Programming Language Model

We assume that we are given a concurrent library written in a *class-based* object-oriented programming language such as `C++` or `Java`. In the following discussion, we introduce the type-based semantics and the concurrency model for such libraries, loosely adhering to the model used in `Java`.

Library and Types. Formally, we define a library $\mathcal{L}$ as a collection of *class* definitions $\langle \mathtt{C}_1, \ldots \mathtt{C}_k \rangle$. Each class $\mathtt{C}_i$ denotes a corresponding *reference type* C_i. A class definition consists of definitions for *data members* (also called fields), and *methods* (member functions). We say that C_2 is a *subtype* of C_1 if $\mathtt{C}_2$ is a subclass of $\mathtt{C}_1$[3].

Data members have *primitive types* (`int`, `double`, *etc.*), or reference types[4]. An object is an instance of a class $\mathtt{C}_i$, and the type of the object is the corresponding reference type C_i. Let $V = \{\mathtt{ob}_1, \ldots, \mathtt{ob}_k\}$ be a (super-)set of all the object variables (*references* in `Java` terminology) occurring in the methods of interest in $\mathcal{L}$.

Access Expressions. Given a universe of object variables V, *access expressions* are constructed as follows:

(a) A variable $\mathtt{ob}_i$ is an access expression of type C_i.

(b) Let e_j be an access expression of non-primitive type C_j, and f_k be a field of the class C_j of type C_k. Then $\mathtt{e} : e_j.f_k$ is an access expression of type C_k.

[3]A subclass $\mathtt{C}_2$ of a class $\mathtt{C}_1$ is defined in standard fashion as a class that contains all definitions of methods and fields of $\mathtt{C}_1$, with possible additional definitions. In other words, subclassing defines an inheritance hierarchy between classes.

[4]Apart from class types, array types are also classified as reference types, and our technique handles array variables conservatively; we omit a detailed discussion on array types for simplicity.

Informally, access expressions are of the form $\texttt{ob}.f_1.f_2.\ldots.f_k$ for some valid sequence of field accesses $f_1, \ldots, f_k$. Let $\mathsf{Type}(e)$ denote the type of an access expression e. A *runtime environment* associates a set of concrete memory locations and values to each object instance and its fields.

Aliasing, Sharing. Aliasing is a relationship between access expressions such that two access expressions e_1 and e_2 are aliased under runtime environment R, if they refer to the same object instance. Two objects $\texttt{ob}_i$, $\texttt{ob}_j$ are said to *share* in a runtime environment R if some access expression of the form $\texttt{ob}_i.f_1.\ldots.f_k$ aliases another expression of the form $\texttt{ob}_j.g_1.\ldots.g_j$. In a type system similar to `Java`'s type system, we can generally assume that if e_1 and e_2 are aliased, then $C_1 : \mathsf{Type}(e_1)$ is a subtype of $C_2 : \mathsf{Type}(e_2)$ or vice-versa. In this case, we also assume that if $f_1, \ldots, f_k$ are the common fields between C_1 and C_2, then $e_1.f_i$ aliases $e_2.f_i$ for all $1 \leq i \leq k$.

A method m of class $\mathcal{C}$ is associated with a signature $sig(m)$ that defines the types for the formal parameters of m, and a return type. Each method m is always executed on an object of some type C_i. The object on which the method is executed is referred to as "`this`" within the method body. The method body consists of a sequence of statements, including calls to other member methods of classes within $\mathcal{L}$. The operational semantics of m are defined using a control-flow graph (denoted $cfg(m)$). We define $cfg(m)$ as a tuple (V_c, E_c, S), where V_c is a set of program points, and E_c is a set of edges, each labeled with a unique program statement $s \in S$.

6.2.1 Synchronization Primitives

Lock-based synchronization. We seek to analyze object libraries that support concurrent accesses to their fields and methods. Therefore, we assume that synchronization statements for *lock-acquisition* and *lock-release* are used to provide *mutual exclusion* for shared data. We assume that these statements are of the form `lock(ob)` and `unlock(ob)`, where `ob` is some object variable. A thread executing `lock(ob)` is blocked unless it can successfully acquire the lock associated with `ob`. The statement `unlock(ob)` releases the lock, returning it to the unlocked state.

In certain languages such as `C++/pthread` and C# there is a designated type for lock variables. For instance, the `pthread` library uses the type `pthread_mutex_t` for mutex locks, and the functions `pthread_mutex_lock` and `pthread_mutex_unlock` to implement acquisition and release of mutexes. In such a model, the programmer decides upon a lock variable to protect access to one or more shared data items. It is the programmer's responsibility to ensure that each access to shared data is preceded by necessary lock acquisition and release, and that the locking discipline is uniform. Failure to do so results in low-level data races or high-level atomicity violations.

```
1: public class Foo {
2:    public void method1() {
3:       .  .  .
4:       synchronized (mon) {
5:          .  .  .
6:       }
7:    }
8:    public synchronized void method2 () {
9:       .  .  .
10:   }
11:}
```

Figure 6.1: Monitor Usage

Monitor-based synchronization. Languages like `Java`, use *monitors* to implement synchronization[5]. A monitor object is a special object with built-in mutual exclusion and thread synchronization capabilities. A *monitored region* corresponding to the monitor `mon` is a sequence of statements that begins with the acquisition of `mon`, and ends with the release of `mon`. In `Java`, *any* object can be used as a monitor, and a monitored region is specified with the help of the `synchronized` keyword. For instance, in the example shown in Figure 6.1, Lines 4-6 constitute a monitored region associated with the monitor object `mon`.

`Java` also allows using the keyword `synchronized` in a method signature (Line 8 in Figure 6.1), which makes the entire method a monitored region corresponding to the implicit object (`this`) on which the method is invoked.

In theory, a monitor is associated with two explicit queues, an *entry*

[5]With the addition of the `java.util.Concurrent` library to `Java`, there is now language support for an explicit lock construct.

queue and a *wait queue*. In `Java`, instead of queues, each monitor maintains an *entry set* and a *wait set*. Queues are intended to implement FIFO access to a monitored region; `Java` makes no such guarantees. In the rest of the presentation, we assume the `Java` model, *i.e.*, we have an entry set and a wait set for each monitor. The entry set is used primarily for *mutual exclusion*, while both sets are used in concert for cooperative synchronization.

Mutual Exclusion with Monitors. Let `mon` be a monitor object. When a thread T reaches the beginning of a monitored region for `mon`, it is placed in the entry set of `mon`. T is granted access to a monitored region if no other thread is executing inside it; we say that T *acquires* `mon` when it enters the monitored region. Any other thread T' that reaches the beginning of the monitored region once `mon` is acquired by T, is placed in the entry set for `mon`. Once T leaves the monitored region, we say that T *releases* `mon`. At this time, some (randomly chosen) thread in the entry set is able to acquire `mon`. Essentially, a monitor maintains the invariant that at any given time there is at most one thread inside the monitored region.

In effect, monitors mimic locks: replace the beginning of a monitored region with `lock(mon)`, the end of the monitored region with `unlock(mon)`. One advantage with having a monitored region is that every monitor acquisition has a matching release, and monitors can be acquired and released only in a strictly nested fashion, *i.e.*, if `ob1` is acquired before `ob2`, then the monitors are released in the *reverse* order.

Co-operation with Monitors. Use of signaling allows monitors to implement cooperation between threads. We focus on the *wait-notify* style of monitors used by `Java`. Each monitor is provided with two special methods: `wait` and `notify`. We explain the semantics of `wait` and `notify` methods with the code fragment shown in Figure 6.2.

A thread (say T_1) executing code fragment P acquires the monitor `mon`, at Line `a0`. After executing code-block A_1, in Line `a2`, T_1 executes the `mon.wait()` statement, which has the following effect: (a) release the monitor `mon`, (b) add T_1 to the wait set for `mon`, (c) suspend execution of T_1. Assume that some other thread, say T_2, reaches the beginning of the monitored region in code fragment Q (Line `b0`) after T_1 has executed Line `a0`. T_2 is then placed in the entry set for `mon`. Once T_1 releases `mon` in Line `a2`, T_2 can enter the monitored region, subsequently executing B_1, followed by `mon.notify()`. The effect of the notify statement is to remove any one thread (say T_1) from the wait set for `mon`, and place it in the entry set for `mon`. T_1 cannot resume execution as it is still in the entry set for `mon`, and T_2 "owns" `mon`. Once T_2 executes Lines `b3, b4`, it releases `mon`. Now T_1 may acquire `mon`, and proceed, executing Lines `a3, a4`.

Condition Variables. To contrast with `wait-notify` monitors in `Java`, we briefly discuss signaling-based synchronization in the `pthread` library. A *condition variable* `cv` is a shared resource that is used in conjunction with a mutex lock `l`. The variable `cv` is typically associated with a Boolean-valued

P:	Q:
a0: synchronized (mon) {	b0: synchronized (mon) {
a1: A_1;	b1: B_1;
a2: mon.wait();	b2: mon.notify();
a3: A_2;	b3: B_2;
a4: }	b4: }

Figure 6.2: Wait-Notify Monitors

expression known as the condition. The method `pthread_cond_wait` has two arguments: `cv` and `l`, and its semantics are similar to that of `wait`: unlock mutex `l`, start waiting on variable `cv`, and upon being notified re-acquire mutex `l`. The method `pthread_cond_signal` takes one argument: `cv`, and its semantics are similar to that of `notify`: issue notification to the variable `cv`.

6.3 Deadlock Detection for Concurrent Libraries

We focus on deadlocks arising from circular dependencies in synchronization constructs such as locks and signaling primitives. Languages such as `Java` combine the mutual exclusion provided by locks with the cooperative synchronization provided by signaling primitives into a single *monitor* construct. Here, we use the abstract term lock to mean both specialized lock variables in languages such as `C`, `C++/pthread`, and monitors used for enforcing mutual exclusion in `Java`.

Static and dynamic approaches for deadlock detection construct *lock-acquisition order graphs* that track dependencies between locks for each thread. Lock-order graphs for concurrent threads are then merged, and a cycle in the

resulting graph indicates a possibility of a deadlock. Such approaches are useful for determining the existence of deadlocks in a *closed* system. Moreover, in closed systems, aliasing information is readily available, or can be conservatively approximated with good precision. Analyzing libraries for deadlocks requires some additional effort due to their *open* nature.

Analyzing concurrent libraries for deadlocks has two main aspects: First of all, we wish to identify if, for *any* client, there are library methods that can be concurrently called in a manner that causes a deadlock. This is termed the *deadlockability* problem. Secondly, we wish to use the results of this analysis to search for the existence of deadlocks in a *particular* client that invokes these library methods. Deadlockability analysis was first introduced by Williams et al. [148]. Therein, the authors construct a lock-order graph for each library method. The *types* of syntactic expressions corresponding to object monitors are used as conservative approximations for the may-alias information between these monitors. The authors show that their approach helps in identifying important potential deadlocks; however, their approach is susceptible to false positives, which have to be then filtered using (possibly unsound) heuristics.

We informally introduce deadlockability analysis with the help of an example. We use the `Java` code snippet (shown in Figure 6.3) from the `EventQueue` class in `Java`'s `awt` library. In Lines 5 and 17, the `synchronized (this)` statement has the effect of acquiring a lock on the `this` object. The `nextQueue` variable is a data member of the `EventQueue` class, which is set

```
 1: public class EventQueue {
 2:    EventQueue nextQueue;
 3:    void postEventPrivate (Event e) {
 4:       .  .  .
 5:       synchronized (this) {
 6:          nextQueue.postEventPrivate(e);
 7:       }
 8:       .  .  .
 9:    }
10:    void push (EventQueue eq) {
11:       .  .  .
12:       nextQueue = eq;
13:       .  .  .
14:    }
15:    void wakeup(boolean f) {
16:       .  .  .
17:       synchronized (this) {
18:          nextQueue.wakeup(f);
19:       }
20:       .  .  .
21:    }
```

Figure 6.3: Methods in `java.awt.EventQueue`

by the `push` method (Line 10). By design, the `postEventPrivate` and `wakeup` methods are intended to perform their action on the `EventQueue` instance `this`, on which they are invoked, and then act on `this.nextQueue` (Lines 6 and 18). Consider the case wherein one client thread (say T_1) invokes `a.push(b)`, while another client thread (say T_2) invokes `b.push(a)`. Subsequently, if T_1 invokes `a.postEventPrivate(e)` concurrently while T_2 simultaneously invokes `b.wakeup(true)`, then this may result in a deadlock. This deadlock can manifest itself in real client code, as reported by client developers in [139, 6542185].

Our deadlockability analysis first performs static inspection of the given concurrent library to identify lock-order graphs for each method. The lock-

order graph for the `wakeup` method in Figure 6.3 captures the acquisition of the lock for the `this` object followed by that of the `this.nextQueue` object:

`this.wakeup(..)`:

Similarly, `postEventPrivate` method first acquires a lock on the `this` object followed by the `this.nextQueue` object, yielding an identical acyclic lock-order graph:

`this.postEventPrivate(..)`:

Consider a client that performs concurrent calls to the methods from two different threads on objects: $\texttt{ob}_1, \texttt{ob}_2$:

$$T_1 : \texttt{ob}_1\texttt{.wakeup(true)} \parallel T_2 : \texttt{ob}_2\texttt{.postEventPrivate()}$$

Assuming no other lock acquisitions are made by the threads themselves, no other calls to methods and no aliasing/sharing between the objects, the lock-order graph of the client is as shown in Figure 6.4.

Normally, the two graphs by themselves are acyclic, and the method calls by themselves do not seem to cause an obvious deadlock. However, the lock-order graph above assumes that the objects $\texttt{ob}_{1,2}$ are not aliased/do not share fields. Consider, on the other hand, the scenario wherein the object

$\texttt{ob}_1\texttt{.nextQueue}$ aliases $\texttt{ob}_2$ and $\texttt{ob}_2\texttt{.nextQueue}$ aliases $\texttt{ob}_1$. Under such a scenario, the lock-order graph of Figure 6.4 is modified by fusing the aliased nodes into a single node to obtain the graph depicted in Figure 6.5. This graph clearly indicates the possibility of a deadlock. Furthermore, prior calls to the `push` methods set up the required pattern of aliasing along the lines of [139, 6542185]. It is important to note that techniques that assume a closed system would only generate the lock-order graph shown in Figure 6.4, and would thus miss a potential deadlock.

At a broad level, our techniques for deadlockability analysis provide a practical framework to:

(a) identify potential deadlock situations by *efficiently* considering all feasible aliasing and sharing scenarios between objects at the concurrent call-sites of library methods,

(b) derive an *interface contract* that characterizes safe aliasing patterns for concurrent calls to library methods.

Concretely, our technique synthesizes the aliasing scenario described above. For calls to the `wakeup` and the `postEventPrivate` methods, our

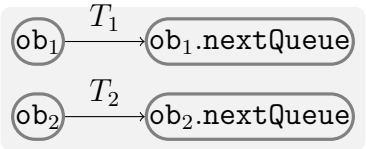


Figure 6.4: Merged Lock-order Graph for `postEventPrivate` & `wakeup`

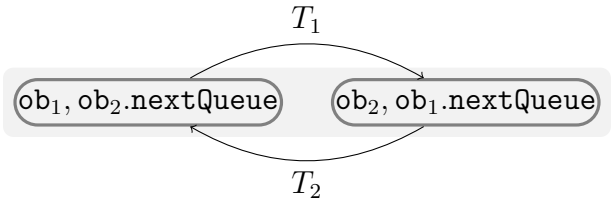


Figure 6.5: Lock-order graph for $T_1||T_2$ under aliasing of nodes.

analysis derives the contract specifying that at any concurrent call to a.`wakeup` () and b.`postEventPrivate` (), the aliasing between `a`,`b` must satisfy:

$$\neg\text{isAliased}(\texttt{a,b.nextQueue}) \ \vee \neg\text{isAliased}(\texttt{b,a.nextQueue})$$

This is sufficient to guarantee deadlock-free execution of these methods assuming that the synchronization operations of the client cannot interfere with that of the library.

In Chapter 7, we elaborate on the steps in deadlockability analysis, and discuss a symbolic encoding that allows us to considerably speed up our technique without loss of precision. In Chapter 8, we extend deadlockability analysis to reason about libraries that use signaling-based synchronization. Experimental evaluation is presented in Chapter 9.

Chapter 7

Symbolic Deadlockability Analysis

In this chapter, we introduce the formal definitions for a lock-order graph, deadlockability analysis, and deadlock-causing aliasing patterns. We then discuss a scheme to encode a lock-order graph into a constraint to enable symbolic reasoning with a SMT-based constraint solver. Finally, we use this scheme to enumerate deadlock-causing aliasing patterns for concurrent libraries, which are then used to derive interface contracts for the library.

7.1 Lock-Graph Computation

Lock-Order Graph. A lock-order graph for method m denoted $lg(m)$ is a tuple (V, E), where V is a set of access expressions, and E is a set of edges. An edge $e_1 \rightarrow e_2$ denotes a pair of nested lock statements `lock(x)` followed by `lock(y)` wherein `x` aliases[1] the access expression e_1, `y` aliases the access expression e_2, and the lock acquisitions are nested along some path in $cfg(m)$ or along a path in the *cfg* of one of m's callees. In what follows, we frequently use the shorter term lock-graph interchangeably with lock-order graph.

[1]For a formal definition of aliasing, please refer Section 6.2.

Procedure computeLG(m)

```
1  begin
2      worklist :=∅
3      V, E, lockset, roots :=∅
4      worklist.push(⊤_m)
5      while (worklist ≠ ∅) do
           /* Edge u -s-> v in cfg(m)  */
6          u :=worklist.deque()
7          succs(u) :={v|(u -s-> v) ∈ cfg(m) }
8          foreach (v in succs(u)) do
9              old_sum :=psum(v)
10             new_sum :=computeFlow(u, s, v)
11             if (old_sum ≠ new_sum) then worklist.push(v)
12             summary(v) :=psum(v) ⊔ new_flow
13     summary(m) :=psum(⊥_m)
14     summaries_map.put(m, summary(m))
15 end
```

Static Forward Lock-Graph Analysis. lock-order graph computation for the methods of a given library involves summarization of each method m within the library. Let $u \xrightarrow{s} v$ be an edge in $cfg(m)$. The partial summary of m at control-flow node v (denoted $\texttt{psum}(v)$) is the symbolic state of m after executing the statement s. It is described as the data structure $(lg(V, E), \texttt{lockset}, \texttt{roots})$, where $lg(V, E)$ is the lock-order graph, `lockset` is the set of locks acquired (but not released) by m at v, `roots` is the set of locks that do not have any incoming edges[2].

We closely follow the technique described in [148] for fixpoint-based summary computation. The procedure computeLG implements a simple work-

[2]In the actual implementation, we also track as a part of the summary a mapping env, that tracks any local variables that may be aliased to global variables on the heap, and thus escape the scope of m. We omit this for simplicity.

list based forward flow analysis. We introduce a single (dummy) entry-point $\top_m$ that has all the actual entry-points of the m as successors, and an exit-point $\bot_m$ that is the successor of all actual exit-points (*i.e.*, `return` statements) for m. We assume that the partially computed summary at each point in the control-flow graph is initialized to $(\emptyset, \emptyset, \emptyset)$. In each step, a new edge $(u \xrightarrow{s} v)$ in $cfg(m)$ is examined. Using the flow equations for the edge as specified by the function computeFlow and the partial summary at control point u ($\texttt{psum}(u)$), we obtain $\texttt{psum}(v)$ (Line 10). If the new $\texttt{psum}(v)$ is different from the original $\texttt{psum}(v)$, then v is added to the work-list (Line 11), and the new $\texttt{psum}(v)$ is *merged* with the old (Line 12). The merge operation ($\sqcup$) computes the union of each component in the partial summary. Finally, the summary of method m (and $lg(m)$ contained therein) is obtained as the merge of the partial summaries at $\bot_m$, which is then stored into a map (Lines 13-14).

We remark that in the presence of recursive (classes containing themselves as members) or mutually recursive types and recursion/loops in the CFG, the fixpoint computation may not terminate, in general. We ensure termination by artificially bounding the length of recursive access expressions in the lock graph nodes.

The computeFlow function is used to compute the effect of a statement s on the partial summary. For the edge $u \xrightarrow{\texttt{lock(mon)}} v$ corresponding to a lock-acquisition, we add edges from every lock $\texttt{mon}'$ in $\texttt{lockset}(u)$ to `mon`, and then add `mon` to $\texttt{lockset}(v)$ (Lines 5-7). The edge $u \xrightarrow{\texttt{unlock(mon)}} v$ corresponding to a lock-release is modeled by setting $\texttt{lockset}(v)$ to the set obtained by

Function `computeFlow`

input : *cfg* edge: $u \xrightarrow{s} v$
output : partial summary *out*

```
1  begin
2      in :=psum(u), out :=∅
       /* in = (V_i, E_i, roots_i, lockset_i), out = (V_o, E_o, roots_o, lockset_o) */
3      switch (s) do
           /* Lock acquisition.  */
4          case lock(mon) :
5              foreach (mon' ∈ lockset_i) do  E_o :=E_i ∪ {(mon', mon)}
6              lockset_o :=lockset_i ∪ {mon}
7              if (roots_i = ∅) then roots_o :={mon}
           /* Lock release.  */
8          case unlock(mon) :
9              lockset_o :=lockset_i − {mon}
10             if (lockset_o = ∅) then roots_o :=∅
           /* Method Invocation.  */
11         case m'(a_1, ..., a_k) :
               /* Check if summary(m') = (V', E', lockset', roots') exists, if
                  not compute it.  */
12             if (summaries_map.contains(m')) then
13                 summary(m') :=summaries_map.get(m')
14             else summary(m') :=computeLG(m')
               /* Map formal parameters in the summary to actual
                  parameters.  */
15             sum' :=summary(m')|_{∀i: f_i ↦ a_i}
               /* Concatenate new summary with current summary.  */
16             E_o :=E_i ∪ E', V_o :=V_i ∪ V'
17             foreach (mon ∈ lockset) do
18                 foreach (mon' ∈ roots') do
19                     E_o :=E_o ∪ {(mon, mon')}
20         otherwise
21             out :=in
22 end
```

removing `mon` from `lockset`(u) (Lines 9-10). Upon encountering a call to a method m', we check if the summary for m' has been computed, and if not, we first compute it. We then replace the formal parameters in `summary`(m') with the actual parameters at the call-site, and concatenate the result sum' with the partial summary computed thus far (Lines 13-19). Concatenation involves adding edges from every lock in `lockset` to every root in the `roots`$'$ (in sum') and adding all other edges in lg' (in sum') to the lock-graph in the partial summary.

7.2 Deadlockability Analysis

Let $m_1, \ldots, m_k$ be a set of methods in library $\mathcal{L}$ that are concurrently invoked by k separate threads. For ease of exposition, we consider the case of two threads (*i.e.*, $k = 2$). However, our results readily extend to *arbitrary* values of k. Let objects $\mathtt{ob}_1, \ldots, \mathtt{ob}_k$ denote a set of objects on which the methods $m_1, \ldots, m_k$ are invoked. Let $\mathtt{ob}_{k+1}, \ldots, \mathtt{ob}_n$ be the set of parameters to these method calls. Let $lg(m_1)$ and $lg(m_2)$ be the lock order graphs for the methods m_1 and m_2 after substituting the `this` object and the formal parameters in m_1 and m_2 with $\mathtt{ob}_1, \ldots, \mathtt{ob}_n$. We assume that $lg(m_1)$ and $lg(m_2)$ are themselves cycle free[3]. Let $V_{1,2} = \{\mathtt{e}_1, \ldots, \mathtt{e}_m\}$ denote the set of access expressions occurring in $lg(m_1)$ or $lg(m_2)$. We first characterize the patterns of aliasing/sharing between the access expressions corresponding to

[3]This assumption relies on re-entrancy of locks. `Java` monitors are re-entrant. Mutexes in `C/C++` with the `pthread` library are commonly defined to be re-entrant. Thus this assumption generally holds true for the libraries that we seek to analyze.

$\texttt{ob}_1, \ldots, \texttt{ob}_n$ under some fixed runtime environment R.

Def. 7.2.1 (Aliasing Pattern)**.** An aliasing pattern α over a set of access expressions V is a symmetric, reflexive and transitive relation over V. If $(e_1, e_2) \in \alpha$ then $(e_1.f_i, e_2.f_i) \in \alpha$ for all shared fields f_i between $\mathsf{Type}(e_1)$ and $\mathsf{Type}(e_2)$.

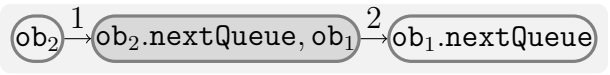
Given graphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$, we use $G = G_1 \sqcup G_2$ to denote the union of the two graphs (*i.e.*, the set of vertices of G is $V_1 \cup V_2$, and the set of edges is $E_1 \cup E_2$). For lock-graphs $lg(m_1)$ and $lg(m_2)$, we refer to $lg(m_1) \sqcup lg(m_2)$ as the merged lock-graph for m_1 and m_2. Given an aliasing pattern α over the nodes of a merged graph $lg(m_1) \sqcup lg(m_2)$, we *fuse* the nodes e_i, e_j of the graph if $(e_i, e_j) \in \alpha$. The outgoing and incoming edges to the individual nodes e_i, e_j are preserved by the fused node. Let $\alpha \triangleright G$ denote the resulting graph after merging all aliased nodes.

Def. 7.2.2 (Deadlock Causing Pattern)**.** An aliasing pattern α is potentially *deadlock-causing* for m_1, m_2 iff $\alpha \triangleright (lg(m_1) \sqcup lg(m_2))$ contains a cycle. An aliasing pattern that is not deadlock-causing is termed *safe*.

Example 7.2.1. Consider two methods from the `java.awt.EventQueue` class: m_1 (`wakeup`) and m_2 (`postEventPrivate`), shown in Figure 6.3. Section 6.3 illustrates the individual lock-order graphs $lg(m_1)$ and $lg(m_2)$. Following the notation established, let $\texttt{ob}_1, \texttt{ob}_2$ denote the objects on which methods m_1, m_2 are invoked, respectively. The access expressions involved in the lock graph $G : lg(m_1) \sqcup lg(m_2)$ are $V_{1,2} = \{\texttt{ob}_1, \texttt{ob}_2, \texttt{ob}_1\texttt{.nextQueue}, \texttt{ob}_2\texttt{.nextQueue}\}$. Let

α_1 be the aliasing pattern $\{(\texttt{ob}_1, \texttt{ob}_2\texttt{.nextQueue})\}$. The merged lock graph $\alpha_1 \triangleright G$ is shown below:

$\alpha_1 \triangleright G$:



The pattern α_1 does not cause a deadlock. However, the following pattern α_2, considered in Section 6.3 is deadlock-causing:

$$\alpha_2 : \{(\texttt{ob}_2, \texttt{ob}_1\texttt{.nextQueue}), (\texttt{ob}_1, \texttt{ob}_2\texttt{.nextQueue})\}$$

Def. 7.2.3 (Deadlockable)**.** A library is termed potentially *deadlockable* if there exists a pair of methods m_1, m_2, and some aliasing pattern α amongst the access expressions in $V_{1,2}$ such that $\alpha \triangleright (lg(m_1) \sqcup lg(m_2))$ contains a cycle.

A simplistic approach consists of (a) enumerating all possible aliasing patterns α, and (b) checking every graph $\alpha \triangleright G$ for a cycle. As pointed out in [148], there may exist a huge number of aliasing/sharing relationships between the parameters, invoked objects, and their fields. Explicit reasoning over such a large number of patterns is intractable, as enumerating all deadlock-causing aliasing patterns is NP-complete, as we show later in Section 7.3. Hence, we use a symbolic representation to encode the graphs and the aliasing patterns as constraints, enabling the use of SAT-Modulo Theory (SMT) solvers to perform the enumeration efficiently.

7.2.1 Symbolic Encoding

We first discuss how we can encode the cycle detection problem into an efficient theory amenable to a SMT solver. The inputs to this problem are a graph $G = lg_1 \sqcup lg_2$, and a fixed aliasing pattern α. In Section 7.3 we will use this encoding to efficiently enumerate all possible patterns to detect potential deadlocks and derive interface contracts.

The overall strategy consists of two parts: We first encode a lock graph G over a set of access expressions V_G as a logical formula $\Psi(G)$. Next, we show how a given alias pattern α may be encoded as a formula $\Psi(\alpha)$. As a result, we guarantee that $\Psi(\alpha) \wedge \Psi(G)$ is unsatisfiable if and only if $\alpha \triangleright G$ has a cycle. The formula $\Psi(G)$ represents a topological ordering of the graph and $\Psi(\alpha)$ places equality constraints on the vertex numbers based on aliasing. If the result is unsatisfiable then no topological order can exist, indicating a cycle.

Graph Encoding. Corresponding to each node $v_i \in V$, we create an integer variable $x(v_i)$ representing its rank in a topological ordering of the node v_i. Corresponding to each edge $v_i \rightarrow v_j$ in the graph, we add the constraint $x(v_i) < x(v_j)$. The resulting formula $\Psi(G)$ is the conjunction of all edge inequalities:

$$\Psi(G) : \left[\bigwedge_{(v_i, v_j) \in E} (\mathbf{x}(\mathbf{v_i}) < \mathbf{x}(\mathbf{v_j})) \right] . \tag{7.2.1}$$

Example 7.2.2. Consider once again the running example from Figure 6.3, continuing with the notation established in Ex. 7.2.1. The merged lock graph $G : \ lg(m_1) \sqcup lg(m_2)$ is recalled in Figure 6.4. The constraint $\Psi(G)$ for this graph is as follows:

$$(x(\mathtt{ob}_1) < x(\mathtt{ob}_1.\mathtt{nextQueue})) \ \wedge \ (x(\mathtt{ob}_2) < x(\mathtt{ob}_2.\mathtt{nextQueue}))\,.$$

Aliasing Pattern Encoding. Given an aliasing pattern α, we wish to derive a formula $\Psi(\alpha, G)$ whose satisfiability indicates the absence of a cycle in $\alpha \triangleright G$ (and conversely). This is achieved by encoding α by means of a set of equalities as follows:

$$\Psi(\alpha) : \ \left[\bigwedge_{(\mathtt{e}_i, \mathtt{e}_j) \in \alpha} (\mathbf{x}(\mathtt{e_i}) = \mathbf{x}(\mathtt{e_j})) \right] . \tag{7.2.2}$$

In effect, the rank of the access expressions that are aliased is required to be the same in the topological order.

Example 7.2.3. Continuing with Ex. 7.2.2, the aliasing pattern

$$\alpha_1 : \{(\mathtt{ob}_2, \mathtt{ob}_1.\mathtt{nextQueue})\}$$

may be encoded as:

$$\Psi(\alpha_1) : (x(\mathtt{ob}_2) = x(\mathtt{ob}_1.\mathtt{nextQueue}))\,.$$

Given an aliasing pattern α, and a graph G, the formulae $\Psi(G)$, $\Psi(\alpha)$ are conjoined into a single formula $\Psi(\alpha, G) : \Psi(G) \wedge \Psi(\alpha)$ that enforces the

requirements for a topological order specified by G, as well as for merging nodes according to the aliasing pattern α.

Example 7.2.4. Continuing with Ex. 7.2.3, recall $\Psi(G)$ from Ex. 7.2.1, consider the combined formula:

$$\Psi(\alpha_1, G): \ (x(\texttt{ob}_1) < x(\texttt{ob}_1.\texttt{nextQueue})) \ \wedge \ (x(\texttt{ob}_2) < x(\texttt{ob}_2.\texttt{nextQueue})) \ \wedge \\ (x(\texttt{ob}_2) = x(\texttt{ob}_1.\texttt{nextQueue}))$$

This formula is satisfiable in the theory of integers, indicating a topological ordering over $\Psi : \alpha_1 \triangleright G$, thus showing that no cycle exists in $\alpha_1 \triangleright G$. On the other hand, consider the formula $\Psi(\alpha_2, G)$ obtained from the pattern:

$$\alpha_2 : \{(\texttt{ob}_2, \texttt{ob}_1.\texttt{nextQueue}), (\texttt{ob}_1, \texttt{ob}_2.\texttt{nextQueue})\}$$
$$\textit{i.e.,} \ \ \Psi(\alpha_2): \ (x(\texttt{ob}_2) = x(\texttt{ob}_1.\texttt{nextQueue})) \ \wedge \ (x(\texttt{ob}_1) = x(\texttt{ob}_2.\texttt{nextQueue}))$$

The combination of $\Psi(G) \wedge \Psi(\alpha_2)$ is clearly unsatisfiable indicating that $\alpha_2 \triangleright G$ has a cycle, which in turn shows that α_2 may cause a deadlock.

Theorem 7.2.1. *The formula $\Psi(\alpha, G)$ is* satisfiable *iff $\alpha \triangleright G$ does not have a cycle.*

Proof. We begin by simplifying the statement of the theorem. Let $G' = \alpha \triangleright G$. Let $\Psi(G')$ be the encoding for G' as per (7.2.1). We observe that $\Psi(G')$ can be obtained from $\Psi(\alpha, G)$ by by replacing integer variables $x(v_i)$ and $x(v_j)$ by a new variable x_{ij}, if $\Psi(\alpha)$ contains the relation $x(v_i) = x(v_j)$. Note that $\Psi(G')$

is satisfiable iff $\Psi(\alpha, G)$ is satisfiable. Thus, we now wish to prove that $\Psi(G')$ is satisfiable iff G' is acyclic.

We first prove that if G' is acyclic, $\Psi(G')$ obtained as per (7.2.1) is satisfiable. Note that the edge relation of an acyclic graph $G'(V, E)$ defines a strict partial order[4] on the set of its vertices, and for a given strict partial order (E) we can define the linear extension of E (denoted E_{tot}) by the order-extension principle. By definition, E_{tot} is a total order, and if $(u, v) \in E$, then $(u, v) \in E_{tot}$. Since E_{tot} is a total order, we can define a bijection f from the set of vertices V to $\mathbb{N}$ such that if $(u, v) \in E_{tot}$, $f(u) < f(v)$. Thus, the interpretation of $\Psi(G')$ where each $x(u)$ is replaced by $f(u)$ evaluates to $true$, *i.e.*, $\Psi(G')$ is satisfiable.

To prove the reverse direction, we prove by contradiction. Assume that $\Psi(G')$ is satisfiable and $G'(V, E)$ contains a cycle. Since $\Psi(G')$ is satisfiable, we can find a satisfying assignment to $\Psi(G')$ such that each $x(v_i)$ corresponds to a distinct integer. By definition, each constraint $x(v_i) < x(v_j)$ corresponds to an edge $(v_i, v_j) \in E$. Since G' contains a cycle, it contains a path $\pi = (v_1, \ldots, v_k, v_1)$ in G', s.t. each consecutive pair of vertices in π is in E. The conjunction of constraints corresponding to π contains the inequality $(x(v_k) < x(v_1))$ and by transitivity of $<$ over integers, also contains $(x(v_1) < x(v_k))$. This is a contradiction as each $x(v_i)$ is a distinct integer. □ □

[4] A strict partial order is a transitive and asymmetric binary relation on a set.

Constraint Solving. Given an aliasing pattern α, the constraint $\Psi(\alpha)$ is a conjunction of equalities, whereas $\Psi(G)$ is a conjunction of inequalities of the form: $v_i < v_j$, *i.e.*, a *unit two variable per inequality* (UTVPI) constraint [98]. In practice, solving Boolean combinations of UTVPI and equality constraints can be solved quite efficiently using modern SMT solvers such as Yices and Z3 [47, 37].

We also note that the problem of solving a set of UTVPI constraints is equivalent to cycle detection in a graph. Therefore, our reduction in this section has not gained/lost in complexity. On the other hand, encoding the graph cycle detection problem as a UTVPI constraint in an SMT framework allows us to efficiently make use of strategies such as *incremental cycle detection* and *unsatisfiable cores*. The subsequent section shows the use of these primitives to effectively enumerate all aliasing patterns by computing *subsumed* and *subsuming* patterns. The discovery of such patterns reduces the set of aliases to be examined and speeds up our approach enormously.

7.3 Contract Generation

We now consider the problem of enumerating all possible aliasing patterns, in order to generate interface contracts for a given concurrent library. The number of such patterns is exponential in the number of nodes of the lock-order graphs. Following Section 7.2, we need to enumerate all possible equivalence classes over the sets of nodes in the lock-order graphs. A naive approach thus suffers from an exponential blow-up. We avoid this using var-

ious optimizations. These optimizations are instances of branch and bound procedures, because in each step large subsets of candidate aliasing patterns that do not cause a deadlock are discarded.

(a) We prune the lock-order graphs to remove all nodes that cannot contribute to a potential deadlock.

(b) We restrict the possible aliasing patterns with the help of a prior alias analysis and typing rules imposed by the underlying programming language.

(c) Based on the set of aliasing patterns already enumerated, we remove sets of *subsumed* or *subsuming* aliasing patterns from consideration.

7.3.1 Pruning Rules

Let E_i, V_i represent the edges and vertices of the lock graph $G_i : \ lg(m_i)$. This pruning strategy is based on the observation that nested lock acquisitions are relatively uncommon and non-nested lock acquisitions may be removed from the lock graph. As a results, nodes without any successors and predecessors can be trivially removed. This results in a large reduction in the size.

A *terminal node* in the graph is defined as one without any successors. Similarly a node in the lock graph is termed *initial* if it has no predecessors.

In general, a terminal node $\mathsf{e}_i \in V$ *cannot* be removed without missing any potential deadlocks.

Example 7.3.1. Returning to the lock graph in Figure 6.4 we note that the two terminal nodes may not be removed since their incoming edges can be used in a potential cycle. The same consideration applies to initial nodes.

However, a terminal node *can* be removed if all the other nodes to which it *may alias* to are also terminal. Similarly, an initial node can be removed if all the other nodes to which it may alias are also initial. The pruning strategy for removing terminal/initial nodes of the graph utilizes the result of a conservative may-alias analysis. Let $\mathsf{mayAlias}(v) = \{u \in V |\ u \text{ may-alias } v\}$.

1. Let v be a terminal node such that all nodes in $\mathsf{mayAlias}(v)$ are also terminal. We remove the vertices in $\mathsf{mayAlias}(v)$ from the graph.

2. Let u be an initial node such that all nodes in $\mathsf{mayAlias}(u)$ are also initial. We remove all nodes in $\mathsf{mayAlias}(u)$ from the graph.

The removal of a terminal/initial node from the graph may create other terminal/initial nodes respectively. Hence, we iterate steps 1 and 2 until no new nodes can be removed. The mayAlias relationship can be safely approximated in languages like Java by *type-masking*. As a result, we regard two nodes as aliased for the purposes of lock graph pruning, if the types of their associated access expressions are compatible (one is a sub-type of another). Note that no potential deadlocks are lost in this process. Our experiments

indicate that the pruned lock graph is an order of magnitude smaller than the original graph obtained from static analysis, making this an important step in making the overall approach scalable. We now shift our focus to reducing the number of aliasing patterns to be enumerated.

7.3.2 Reducing Aliasing Patterns

Given two graphs with n nodes each, the number of possible aliasing patterns that need to be considered across the nodes of the two graphs is exponential in n. In our experiments with `Java` libraries, we have observed that due to the extensive use of locking with the `synchronized` keyword lock-order graphs contain 100s of nodes. After pruning, we reduce these to lock-order graphs with 10s of nodes. However, given the exponential number of aliasing patterns that may exist, we need to impose restrictions on the set of aliasing patterns that we examine. First of all, it suffices to consider aliasing patterns that respect the type safety considerations of the language and the conservative may-alias relationships between nodes.

Def. 7.3.1 (Admissible)**.** An aliasing pattern α is admissible iff for all $(u, v) \in \alpha$, $u \in \mathsf{mayAlias}(v)$. Once again, type information can be used in lieu of alias information for languages such as `Java`.

Another important consideration for reducing the aliasing patterns, is that of *subsumption*. Subsumption is based on the observation that for a deadlock causing pattern α adding more aliases to α does not remove the

deadlock. Similarly, for a safe pattern β, removing aliases from β does not cause a deadlock.

Def. 7.3.2 (Subsumption)**.** A pattern α_2 *subsumes* α_1, denoted $\alpha_1 \subseteq \alpha_2$, iff $\forall(u,v) : (u,v) \in \alpha_1 \Rightarrow (u,v) \in \alpha_2$. In other words, α_1 is a sub-relation of α_2.

Lemma 7.3.1. *If α_1, α_2 are aliasing patterns, and $\alpha_1 \subseteq \alpha_2$, then the following are true:*

(A) α_1 is deadlock-causing $\Rightarrow$ α_2 is deadlock-causing,

(B) α_2 is safe $\Rightarrow$ α_1 is safe.

Proof. Since $\alpha_1 \subseteq \alpha_2$, $\Psi(\alpha_2, G)$ can be expressed as $\Psi(\alpha_1, G) \wedge \Psi(\alpha_2 \backslash \alpha_1)$. Since we know that α_1 is deadlock-causing, by definition $\Psi(\alpha_1, G)$ is unsatisfiable. As the conjunction with an unsatisfiable formula is unsatisfiable, it follows that $\Psi(\alpha_2, G)$ is unsatisfiable. □

Note that (B) is simply the contrapositive of (A) and is stated here in Lemma 7.3.1 for the sake of exposition.

Def. 7.3.3 (Maximally Safe/Minimally Unsafe)**.** A pattern α that causes a deadlock is *minimally unsafe* iff for any $(u,v) \in \alpha$, $\alpha \setminus \{(u,v)\}$ is not deadlock causing. Similarly, a safe (non-deadlock) pattern α is maximally safe if, for any $(u,v) \notin \alpha$, $\alpha \cup \{(u,v)\}$ is deadlock causing.

Following Lemma 7.3.1, it suffices to enumerate only the maximally safe and minimally unsafe patterns. Hence, after enumerating a pattern α

that is *safe*, we can add previously unaliased pairs of aliases to α as long as the addition does not cause a deadlock. The resulting pattern is a *maximally safe pattern*. Similarly, upon encountering a deadlock-causing pattern β, we remove "unnecessary" alias pairs from β as long as pairs that contribute to some cycle in $\beta \triangleright G$ can be retained.

Example 7.3.2. Consider the aliasing pattern $\alpha_0 : \emptyset$ for the example described in Section 6.3. Figure 6.4 shows the resulting graph. We can add the pair $(\texttt{ob}_1, \texttt{ob}_2.\texttt{nextQueue})$ to α_0 without creating any cycles. The resulting pattern α_1 is shown in Ex. 7.2.1. However, if we add the pair $(\texttt{ob}_2, \texttt{ob}_1.\texttt{nextQueue})$ to α_1 then we obtain a cycle in the graph. As a result, the pattern α_1 is maximally safe.

The explicit enumeration algorithm (Algorithm 7.3) for aliasing patterns maintains a set U of unexplored patterns, sequentially exhausting the unexamined patterns from this set while updating the set U. The algorithm terminates when $U = \emptyset$. First of all, a previously unexamined pattern α is chosen from the set U (Line 4), and the graph $\alpha \triangleright G$ is examined for a cycle (Line 5). If the graph is acyclic, we keep adding previously unaliased pairs (u, v) to α as long as the addition does not create a cycle in $\alpha' \triangleright G$, where α' is the symmetric and transitive closure of $\alpha \cup \{(u, v)\}$. The result is a pattern α that is maximally safe, which is then added to the set $\mathcal{S}$ (Line 9). We then remove all patterns β that are subsumed by α from the graph G, as they are safe (Line 10). On the other hand, if the graph $\alpha \triangleright G$ has cycles, we

Algorithm 7.3: EnumerateAllAliasingPatterns

```
   Input: G : Graph
   Result: 𝒟 : Deadlock Scenarios
 1 begin
 2     U:= all legal aliasing patterns
 3     while U ≠ ∅ do
 4         Choose element α ∈ U.
 5         if α ▷ G is acyclic then
               /* Add aliases without creating a cycle */
 6             foreach (u, v) ∉ α do
                   /* Add (u, v) and compute closure.  */
 7                 α′ := Closure({(u, v)} ∪ α)
 8                 if α′ ▷ G is acyclic then α := α′
               /* α maximally safe */
 9             𝒮 := 𝒮 ∪ {α}
10             U := U \ {β | β ⊆ α}
11         else                                   /* α ▷ G has a cycle */
               /* Choose a cycle C */
12             C:=FindACycle(α ▷ G)
               /* Remove aliases that do not contribute to C */
13             α′:=α ∩ {(u, v) | u, v ∈ C}
               /* α′ is unsafe  */
14             U := U \ {β | α′ ⊆ β}
15             𝒟 := 𝒟 ∪ {α′}
16
17 end
```

choose some cycle C in the graph (Line 12), and the aliases in α that involve the merged nodes in C. Discarding all the *superfluous* aliases not involving nodes in the cycle C yields an alias relationship $\alpha' \subseteq \alpha$ that is still deadlock-causing [5] (Line 13). The set U of unexamined patterns is pruned by removing all patterns that subsume α' (such patterns also cause a deadlock) (Line 14).

The application of Algo. 7.3 on the graph from Figure 6.4 enumerates

[5]Note that α' may not be a minimally unsafe relation.

the max. safe/ min. unsafe patterns in Table 7.1.

Table 7.1: Max. Safe/Min. Unsafe Patterns Enumerated.

$\{(\texttt{ob}_1, \texttt{ob}_2), (\texttt{ob}_1.\texttt{nextQueue}, \texttt{ob}_2.\texttt{nextQueue})\}$	SAFE
$\{(\texttt{ob}_1, \texttt{ob}_2.\texttt{nextQueue})\}$	SAFE
$\{(\texttt{ob}_2, \texttt{ob}_1.\texttt{nextQueue})\}$	SAFE
$\{(\texttt{ob}_1, \texttt{ob}_2.\texttt{nextQueue}), (\texttt{ob}_2, \texttt{ob}_1.\texttt{nextQueue})\}$	DL

Symbolic Enumeration Algorithm. Algorithm 7.3 relies on explicit representation of the set U of alias patterns in order to perform the enumeration. Representing an arbitrary set of relations explicitly is not efficient in practice. Therefore, we leverage the power of symbolic solvers to encode aliasing patterns succinctly. Specifically, we wish to represent the set U of unexamined aliasing patterns with the help of a logical formula. Let $V = \{\mathsf{e}_1, \ldots, \mathsf{e}_k\}$ be the set of access expressions labeling the nodes of the graph G. We introduce a set of integer variables $y(\mathsf{e}_i)$, such that each $y(\mathsf{e}_i)$ corresponds to an access expression e_i. We then encode all aliasing patterns with the help of a logical formula Ψ_0 involving the $y(\mathsf{e}_i)$ variables, as follows:

$$\Psi_0(V) = \forall_{e_i, e_j \in V} \left[\begin{array}{l} \bigwedge_{e_i \notin \mathsf{mayAlias}(e_j)} (y(e_i) \neq y(e_j)) \wedge \\ \bigwedge_{e_i.f, e_j.f \in V} ((y(e_i) = y(e_j)) \Rightarrow (y(e_i.f) = y(e_j.f))) \end{array} \right]$$

The formula Ψ_0, ensures the consistency of alias patterns considered in the enumeration process. Specifically, expressions that cannot be aliased to each other according to a conservative pointer analysis are not considered

aliased in any of the patterns generated. Secondly, if e_1, e_2 are aliased then for every field f, $e_1.f$ and $e_2.f$ must be aliased (provided the two expressions are in the set V). Algorithm 7.4 shows the symbolic version of Algorithm 7.3. The correspondence between the two algorithms is immediately observable upon comparing them. Since we represent sets of aliasing patterns as a logical formula, a witness to the satisfiability of this formula is an aliasing pattern α (Line 5).

Recall from Section 7.2.1 that we can encode the problem of cycle detection in a graph using inequality constraints. Thus, in Line 12 we check the inequality constraints specified by the graph G, *i.e.*, $\Psi(G)$, conjoined with the previously unexamined aliasing pattern α (encoded as $\Psi(\alpha)$) for satisfiability. Satisfiability of this formula indicates that the graph G is cycle-free, and we proceed to compute a maximally safe aliasing pattern from the given α (Line 10). Once a maximally safe α is obtained, we remove all aliasing patterns that are subsumed by α from the set of all aliasing patterns (represented by Ψ_U), and add α to $\mathcal{S}$ (Line 12). If the formula is unsatisfiable, then we obtain the minimal unsatisfiable core (Line 14) and extract the minimally unsafe aliasing pattern α' from the constraints represented in this core. We then remove all aliasing patterns that subsume α' from Ψ_U (Line 16), and add the minimally unsafe α' obtained (if any) to the set $\mathcal{D}$ (Line 17).

Such a symbolic encoding of sets of aliasing patterns has many advantages, including: a) the power of constraints to represent sets of states compactly, and b) the use of blocking clauses to remove a set of subsumed

Algorithm 7.4: SymbolicEnumerateAllAliasingPatterns

```
   Input: G : Graph
   Result: 𝒟 : Deadlock Scenarios
 1 begin
 2     Ψ_U:=Ψ_0(V) (encoding all alias patterns)
 3     while Ψ_U SAT do
 4         (y(e_1), ..., y(e_k)) := Solution of Ψ_U.
 5         α:={(e_i, e_j) | y(e_i) = y(e_j)}.
           /* Construct Ψ(α, G)  */
 6         if Ψ(α, G) SAT then
               /* Add aliases without creating a cycle */
 7             foreach (e_i, e_j) ∉ α do
 8                 α':=Closure(α ∪ (e_i, e_j))
 9                 Ψ(α', G):=Ψ(α, G) ∧ (x(e_i) = x(e_j))
10                 if Ψ(α', G) SAT then α := α'
               /* α is maximally safe */
11             Ψ_U:=Ψ_U ∧ ⋁_{(e_i,e_j)∉α} y(e_i) = y(e_j)
12             𝒮 :=𝒮 ∪ {α}
13         else
               /* Ψ(α, G) UNSAT */
14             C:=MinUnsatCore(Ψ(α, G))
15             α':={(e_i, e_j) | x(e_i) < x(e_j) constraint in C}
               /* α' is unsafe  */
16             Ψ_U:=Ψ_U ∧ ⋁_{(e_i,e_j)∈α'} y(e_i) ≠ y(e_j)
17             𝒟 := 𝒟 ∪ {α'}
18 end
```

or subsuming aliasing patterns. Modern UTVPI solvers such as YICES and Z3 incorporate techniques for fast and incremental cycle detection upon addition or deletion of constraints [47, 37]. This is very useful in the context of Algorithm 7.4. In practice, our use of subsumption and pruning ensures that a very small fraction amongst the alias patterns is explored by the symbolic algorithm.

7.3.3 Rationale for Symbolic Encoding

We now formally justify the use of a Boolean encoding along with SAT/SMT solvers to perform the symbolic enumeration of unexamined alias patterns. Specifically, we justify the use of SAT to test for unexamined patterns in line 12 of Algorithm 7.4 by showing that the underlying problem of detecting unexamined alias patterns is NP-complete. Let $G_1 : (N_1, E_1)$ and $G_2 : (N_2, E_2)$ be two graphs. An aliasing pattern is a binary relation $\alpha \subseteq N_1 \times N_2$ between the nodes of G_1 and G_2. Recall that the execution of our algorithm for symbolic enumeration of "interesting" aliasing patterns yields the set $\mathcal{S}$ (set of aliasing patterns that are maximally safe) and the set $\mathcal{D}$ (set of aliasing patterns that are minimally unsafe). Also recall that in the set $\mathcal{S}$, maximally safe patterns are obtained by adding aliases to safe patterns as long as they do not cause deadlocks (cf. line 10 in Algorithm 7.4). Similarly, minimally unsafe patterns are added to $\mathcal{D}$ by removing pairs of aliases from a deadlock causing pattern until no more can be removed (cf. line 14 in Algorithm 7.4).

We say that a pattern α is *unexamined* w.r.t. $\mathcal{S}, \mathcal{D}$ iff

$$(\forall S_i \in \mathcal{S}\ \alpha \not\subseteq S_i) \text{ and } (\forall\ D_i \in \mathcal{D}\ D_i \not\subseteq \alpha)\ .$$

We now consider the problem AnyUnexaminedPatterns as below:

Inputs:	$(G_1, G_2, \mathcal{S}, \mathcal{D})$
Output:	YES, iff $\exists \alpha \subseteq N \times N$ *unexamined* w.r.t. $\mathcal{S}, \mathcal{D}$.
	NO, otherwise.

Theorem 7.3.1. *AnyUnexaminedPatterns is NP-complete.*

Proof. Membership in NP is straightforward. An aliasing pattern α claimed to be unexamined can be checked by iterating over the aliasing patterns in $\mathcal{S}$ and $\mathcal{D}$, and checking (in polynomial time) for the subset relation.

We prove NP-hardness by reduction from the CNF satisfiability problem. Let $V = \{x_1, \ldots, x_n\}$ be a set of Boolean-valued variables and $\mathcal{C} = \{C_1, \ldots, C_m\}$ be a set of disjunctive clauses over literals of the form x_i or $\neg x_i$. Corresponding to this instance of SAT, we create an instance $\langle G_1, G_2, \mathcal{S}, \mathcal{D} \rangle$ of the AnyUnexaminedPatterns problem.

Consider a graph G_1 consisting of n vertices, each labeled with a variable in V. Consider a graph G_2 consisting of two vertices labelled *true* and *false*, respectively. Informally, aliasing between the node labeled x_i in G_1 and a node in G_2 can be interpreted as an assignment of *true* or *false* to x_i. We now design the sets $\mathcal{S}$ and $\mathcal{D}$ so that any unexamined aliasing pattern α has the following properties:

1. For each x_i, exactly one tuple in the set $\{(x_i, true), (x_i, false)\}$ belongs to α. In other words, α represents an assignment of truth values to variables in V.

2. The assignment represented by α is a solution to the original SAT problem.

We define the set $\mathcal{S}$ as $\{A_1, \ldots, A_i, \ldots, A_n\}$, where

$$A_i : \ (V \setminus \{x_i\}) \times \{true, false\} .$$

Intuitively, each A_i represents an aliasing pattern in which the x_i variable is missing, and all other variables have both the $true$ and $false$ value assigned. Clearly, any unexamined pattern that is a subset of any A_i does not have a truth-value assigned to the variable x_i, and hence cannot represent a valid assignment of truth values to the original SAT problem. We define the set $\mathcal{D}$ as a union of two sets B and T. The set B is defined as $\{B_1, \ldots, B_n\}$, where:

$$B_i : \ \{(x_i, true), (x_i, false)\} .$$

Intuitively, any aliasing pattern α that is a superset of some B_i cannot represent a valid truth value assignment to the original SAT instance, as it would contain conflicting assignments of truth values to the variable x_i.

The set T is defined in terms of the clauses $C_i \in \mathcal{C}$. $T = \{T_1, \ldots, T_m\}$, wherein T_i corresponds to the i^{th} clause C_i as follows:

$$T_i = \bigcup_j \begin{cases} \{(x_j, false)\} & x_j \in C_i \\ \{(x_j, true)\} & \neg x_j \in C_i \end{cases}$$

Intuitively, any aliasing pattern α that is a superset of T_i cannot satisfy the clause C_i (*i.e.*, $C_i = false$). Hence, such an α cannot represent a solution to the SAT problem. Combining B and T, any α that is the superset of any aliasing pattern $D_i \in \mathcal{D}$, thus, cannot represent a solution to the original SAT problem.

To summarize, corresponding to each SAT instance (V, C), we construct an instance of the AnyUnexaminedPatterns problem with

$$\mathcal{S}: \{A_1, \ldots, A_n\}, \text{ and, } \mathcal{D}: \{B_1, \ldots, B_n\} \cup \{T_1, \ldots, T_m\}$$

In order to complete the proof, we show that there is a satisfying assignment to the original problem if and only if there is an unexamined aliasing pattern.

Let $\mu : \{x_1, \ldots, x_n\} \mapsto \{true, false\}$ be any satisfying solution to the original problem. We construct a pattern α that maps x_i to $true$ if $\mu(x_i) = true$ and to $false$ otherwise. We now show that α is an unexamined aliasing pattern. It is easy to see that $\alpha \not\subseteq A_i$, since α contains at least one of $(x_i, true)$ or $(x_i, false)$. We can also show that $B_i \not\subseteq \alpha$ since α contains only consistent assignments for each variable x_i. Similarly, $T_i \not\subseteq \alpha$, or else the corresponding clause C_i is not satisfied by α. Therefore α is an unexamined aliasing pattern. Conversely, we can demonstrate that any unexamined aliasing pattern α that can be discovered corresponds to a satisfying truth assignment. This shows that CNF-SAT reduces to the problem AnyUnexaminedPattern, and is thus NP-complete. □

7.3.4 Deriving a Contract

The enumeration scheme in Algorithm 7.3 and Algorithm 7.4 can generate a contract that *succinctly* represents the set of all safe aliasing patterns. The result of the enumeration is a set of patterns $\mathcal{D}$ such that any aliasing

pattern β is deadlock-causing iff it subsumes a pattern $\alpha \in \mathcal{D}$.

Lemma 7.3.2. *An aliasing pattern α is safe iff for all $\beta \in \mathcal{D}$, $\beta \not\subseteq \alpha$.*

Proof. We prove this by contradiction. Suppose α is safe, and there exists $\beta \in \mathcal{D}$, s.t. $\beta \subseteq \alpha$. By definition, α subsumes β; hence, if β is deadlock-causing, α should be deadlock-causing, which is a contradiction. □

In practice, contract derivation consists of first *compacting* the set $\mathcal{D}$ to obtain the minimal deadlock-causing patterns. The contract for safe calling contexts can then be expressed succinctly using the fact that any such pattern must not subsume any element of the set $\mathcal{D}$.

Example 7.3.3. From Table 7.1, the only unsafe pattern enumerated is

$$\alpha_2 : \{(\texttt{ob}_1, \texttt{ob}_2.\texttt{nextQueue}), (\texttt{ob}_2, \texttt{ob}_1.\texttt{nextQueue})\}$$

The set of safe patterns therefore is specified by the following set:

$$\left\{\alpha \;\middle|\; (\texttt{ob}_1, \texttt{ob}_2.\texttt{nextQueue}) \notin \alpha \text{ or } (\texttt{ob}_2, \texttt{ob}_1.\texttt{nextQueue}) \notin \alpha \right\}.$$

In terms of a *contract*, this set is expressed as

$$\neg\text{isAliased}(\texttt{ob}_1, \texttt{ob}_2.\texttt{nextQueue}) \;\vee\; \neg\text{isAliased}(\texttt{ob}_2, \texttt{ob}_1.\texttt{nextQueue}).$$

Theorem 7.3.2. *The set $\mathcal{D}$ of deadlock-causing alias patterns for each pair of library methods obtained by the enumeration technique in Algorithm 7.4 yields a* contract *of the form:*

$$\bigwedge_{\alpha \in \mathcal{D}} \bigvee_{(e_i, e_j) \in \alpha} \neg \mathit{isAliased}(e_i, e_j).$$

Note that the contract is a Boolean combination of propositions conjecturing aliasing between access expressions. Thus, such a contract can be both statically and dynamically enforced in a client, as the concrete aliasing information between access expressions can be obtained through alias analysis, or may be available at run-time.

7.4 Analyzing Clients

The interface contracts generated by our tool vastly simplify the analysis of client code that makes use of the library methods that are part of the library's interface contract. Furthermore, they serve to document against the improper use of the methods in a multi-threaded context.

We recall from Section 7.3 that the final contract for a safe call to a pair of methods m_1, m_2 is a Boolean expression involving propositions of the form $\neg\text{isAliased}(\mathsf{e}_i, \mathsf{e}_j)$, wherein e_i and e_j are access expressions corresponding to the formal parameters of the methods, including the "`this`" parameter. In practice, checking such a contract for a given client that uses the library involves two major components: (A) a *May-Happen in Parallel* (MHP) analysis [102] for calls to methods m_1 and m_2 to determine if two different threads may reach these method call-sites simultaneously, and (B) a conservative, thread-safe alias analysis in order to determine the potential aliasing of parameters at the invocation sites of the methods in question.

On the basis of such an alias analysis, we may statically evaluate the contract at each concurrent call-site. Note that these two components are

already part of most data-race detection tools such as *CHORD* [109, 110]. In theory, deadlock violations can be directly analyzed by a "whole-program analysis" of the combined client and the library code. In practice, this requires the (re-)analysis of a significant volume of code. Using contracts has the distinct advantage of being fast in the case of small clients that invoke a large number of library methods. Moreover, decoupling the client analysis from the library analysis allows our technique to be compositional. Since library internals are often confusing and opaque to the client developers, another key advantage is the ability to better localize failures to their causes in the clients, as opposed to causes inside the library code.

We remark that the deadlock analysis in the client is considerably simpler if the client does not use any additional synchronization. In this case, any two statements in each client thread can be conservatively treated as being simultaneously reachable, *i.e.*, *may happen in parallel*. If the two simultaneously reachable statements are library method invocations, then these can cause a deadlock if the aliasing between the client variables corresponds to a deadlock-causing aliasing pattern. In addition to the aliasing information gleaned from the MHP analysis, any aliasing between client variables that a library method may introduce is available as the map env, which is part of the method summary. Thus, given a map Als that contains a set of aliases for each thread-local and global client variable, we append the mappings in env to Als. We then use the computed alias information, and the interface contracts of the library to determine if the aliasing at the respective call-sites

corresponds to a deadlock-causing aliasing pattern.

For clients that use their own synchronization, computing the *may happen in parallel* relation is much harder. Here, we can utilize techniques that can perform a concurrent reachability analysis such as in [93, 109]. For any simultaneously reachable statements s and t, we can obtain the partial summaries as in Section 7.1 and aliasing information from a thread-safe alias analysis. If s and t are calls to library methods, we can use the interface contracts and the aliasing information to first check for possible deadlocks due to calls to the library methods. If not, we can check for any *high-level* deadlocks, *i.e.*, deadlocks due to circular dependencies in the client's synchronization primitives by merging the partial summaries for s and t.

We remark that due to the conservative approximations in computing simultaneous reachability, some reported deadlocks could be spurious. This occurs in the following situations: when threads involved in deadlock meticulously use the control flow to prevent simultaneous reachability of the deadlock-causing method invocations; or when applications use a tree hierarchy for ordering locks that will thus not alias in practice. In our experiments, we have found the latter to be much more common than the former.

To fully verify if a reported deadlock is a true deadlock, we would need to use dynamic techniques like run-time analysis, model checking or testing. Some false positives could be filtered by annotating lock acquisitions with *path expressions*, *i.e.*, the precise conditions in the code that are required to be true, for a particular lock to be acquired. Path expressions can be gleaned from

static analysis of the library code, and evaluated with the help of a suitable constraint solver.

7.5 Bibliographic Notes

Deadlock detection has been a well-studied problem with various extant approaches. We highlight some of the key approaches that merit a comparison. Most of the material in this chapter is drawn from [40].

Runtime Techniques. Runtime techniques for deadlock detection track nested lock acquisition patterns. The *GoodLock* algorithm [79] is capable of detecting deadlocks arising from two concurrent threads; [3] generalizes this to an arbitrary number of threads, and defines a special type system in which potential deadlocks correspond to code fragments that are untypable. Agarwal et al. [2] further extend this approach to programs with semaphores and condition variables.

Model Checking. Model checking techniques [30] have been successfully used to detect deadlocks in programs. For instance, Corbett et al. employ model checking to analyze protocols written in Ada for deadlocks [32]. Model checkers such as *SPIN* [84], *Java Path Finder* [81, 79] have been used extensively to check concurrent `Java` programs for deadlocks. However, program size and complexity limit these approaches in presence of arbitrary aliasing. A compositional technique based on summarizing large libraries can help these

approaches immensely. Bensalem et al. [10] propose a dynamic analysis approach, based on checking synchronization traces for cycles, with special emphasis on avoiding certain kinds of guarded cycles that do not correspond to a realizable deadlock.

Static Techniques. Static techniques based on dataflow analysis either use dataflow rules to compute lock-order graphs [145, 62] or examine well-known code patterns [5, 118] to detect deadlocks. Naik et al. present an interesting combination of different kinds of static analyses to approximate six necessary conditions for deadlock [110]. Most static techniques focus on identifying deadlocks within a given closed program, while in [110], the authors close a given open program (the library) by manually constructing a harness for that program. In [135] the author analyzes the entire `Java` library, and uses a coarser level of granularity in lock-order graph construction.

Deadlock Detection for Libraries. As mentioned previously, deadlock analysis for concurrent libraries was first introduced by Williams et al. [148] for analyzing `Java` libraries. Therein, the authors use types to approximate the may-alias relation across nodes in the lock-order graphs for a library, and reduce checking existence of potential deadlocks to cycle detection. Our approach is inspired by this work and seeks to solve the very same problem under similar assumptions. Our distinct contributions lie in the use of aliasing information in the library. As Williams et al. rightly point out, there is an

overwhelming amount of aliasing possible. Therefore, we use pruning as well as symbolic encoding of the aliasing patterns. Our use of subsumption ensures that a tiny fraction of the exponentially many alias patterns are actually explored, and doing so clearly reduces the number of false positives without the use of unsound filtering heuristics. The use of aliasing pattern subsumption also ensures that the final deadlock patterns can be inverted to yield statically enforceable interface contracts.

Chapter 8

Deadlocks in Signaling-based Synchronization

In Chapter 7, we focussed on deadlocks arising from circular dependencies in lock acquisition. Recall that a lock is really an abstraction for specifying mutual exclusion, implemented as `Java` monitors or `pthread` mutexes. In languages such as `Java`, each object monitor is provided with `wait` and `notify` methods to achieve signaling-based synchronization. In this chapter, we extend the nested-monitors rule used during lock-graph computation in Chapter 7 to account for signaling-based synchronization. We call this extended rule the *generalized nested-monitors rule*. We then show how we can use static analysis to construct extended lock-graphs that encode potential deadlocks. Finally, we discuss a symbolic encoding scheme for such extended lock-graphs.

Background

We briefly recall the semantics of `wait` and `notify` methods. A thread T executing a `mon.wait()` statement inside the monitored region for `mon` has the effect of releasing `mon`, and suspending its execution. A thread T' executing `mon.notify()` has the effect of waking up a thread T that might be waiting (on

`mon`). Upon waking up the thread T tries to re-acquire `mon` before it can resume execution. For precise details, please see Section 6.2.1. As before, we assume that the library is intended to be well-encapsulated: every `wait` statement in some library method is expected to have a matching `notify` statement from within some (possibly the same) library method.

Happens Before. We define a *happens before* relation similar to [100] applied to concurrent systems (cf. [66]). In simple terms, a statement s_1 happens before s_2 (denoted $s_1 \rightarrow s_2$) if causal precedence can be established between the execution of s_1 by T and s_2 by some (possibly same) thread T'. For instance, statements in the same thread are trivially ordered by $\rightarrow$, and causal precedence is established across statements in different threads by synchronization operations such as lock release, lock acquisition, thread notification, *etc.*.

8.1 Generalized Nested-Monitors Rule

In this section, we first elaborate on scenarios that can cause deadlocks in libraries with `wait-notify` statements. We postulate a rule to capture such scenarios in the generalized nested-monitors rules, and show how these rules can be used to construct an extended lock-graph.

8.1.1 Deadlock Scenarios

A thread executing a method that contains a statement $s =$ `mon.wait()` is suspended upon executing s. It is understood that a waiting thread will be eventually notified by another thread that executes the statement $r =$ `mon.notify()`, failing which the waiting thread fails to progress (causing a deadlock). Such a lack of notification can be ascribed to either a *missing*, or a *lost* or an *unreachable* notification. We elaborate on these scenarios as follows:

I. For some `wait()` statement, there is no matching `notify()` statement in the library. This is an instance of a missing notification.

II. For some concurrent execution, it is possible that every notify statement r satisfies that $r \rightarrow s$. This is an instance of a lost notification, *i.e.*, the appropriate object *is* notified, but *before* it has a chance to wait.

III. Assume that for every `wait` statement s, there is a matching `notify` statement r present in some library method (*i.e.*, notify is *not* missing). For some concurrent execution, $s \rightarrow r$, but r is unreachable in a possible notifying method.

Analyzing programs with wait-notify synchronization is hard. In fact, context-sensitive synchronization-sensitive analysis is undecidable [124]. As the general problem is undecidable, we use a case-by-case analysis to identify sub-problems with conservative solutions.

To statically detect Case I, we can make use of a thread-aware, thread-safe alias analysis: For a given pair of methods (such that one invokes a `wait`, and other a `notify`), we need to check if the `wait` and `notify` are invoked on objects that alias to each other. We remark that that this case subsumes the common beginner mistake of `Java` programmers to invoke `wait` and `notify` on the `this` object from within two different monitors. The result is that the `wait` is issued on one monitor, while the `notify` is issued on an entirely different monitor, causing the waiting thread to wait forever.

Case II requires semantic analysis of the code to deduce whether there is an interleaving in which a `notify` precedes the matching `wait` statement. Thus, case II is tricky to detect statically without introducing a slew of false positives. It may be possible to predict such deadlocks by conservative approximation of the happens before relation; however, this is beyond the scope of this paper.

Case III can manifest due to different reasons, such as (a) the `notify` is in a control-flow path that the "notifying" method does not execute, (b) waits and notifies are mismatched, and (c) methods acquire monitors in a nested fashion. As in Case II, (a) requires semantic analysis, and goes beyond the scope of this paper. Examples of Case III(b) include cases where there are more `wait` statements that `notify` statements, and there is a circular dependency between `wait` and `notify` instructions,*i.e.*, T_1 executes `mon1.wait()` followed by `mon2.notify()`, while T_2 executes `mon2.wait()` followed by `mon1.notify()`. We can extend our approach to statically detect these kind of deadlocks by

checking compatibility between `wait-notify` sequences as a part of future work. In what follows we focus on Case III(c).

8.1.2 Nested Monitor Deadlocks

Nested monitor deadlocks have been well-known in the literature since before the advent of programming languages that use monitors, cf. [103]. We wish to analyze potential nested monitor deadlocks in concurrent libraries written in languages such as `Java` that use `wait-notify` constructs. As before, we assume that library methods are invoked by separate threads, and use the terms threads and methods interchangeably.

The semantics of `wait-notify` create situations that not only lead to unpleasant deadlocks, but also violate the well-encapsulation principle of modular software. We explain this with an example.

Example 8.1.1. The library method `foo` contains a `this.wait()` statement within the monitored region for `this`. `foo` is invoked on the object `lib` from within the monitored region for `mon1` inside the client method `bar`. As the acquisition of `this` (i.e. `lib`) is nested within the scope of `mon1`, this is an instance of a nested monitor acquisition. Recall the semantics of `wait` from Section 6.2.1. For a thread T executing the `this.wait()` statement, the effect is that T releases the monitor associated with `this`, but not the monitor `mon1`. Thus, a library method `foo` holds on to a client resource, which violates the spirit of well-encapsulation.

```
1: public class Library {
2:    public synchronized void foo() {
3:       this.wait();
4:    }
5: }
```

```
1: public class Client {
2:    Library lib;
3:    public void bar() {
4:       synchronized (mon1) {
5:          lib.foo();
6:       }
7:    }
8: }
```

Thus, we argue that among all the different cases in which deadlocks manifest in concurrent libraries using `wait-notify`, these kind of deadlocks are the most important to document and predict. We now give concrete examples of nested monitor deadlocks due to `wait-notify`.

Unreachable Notification. Consider the case where a method m acquires some monitors before executing the `mon.wait()` statement. As per the semantics of `wait()`, (the thread executing) m releases `mon`, but while waiting, it still *holds* the previously acquired monitors. Now, any method m' that needs to acquire one of the "held" monitors before it can execute s_2 (`mon.notify()`) will never reach s_2. Thus, method m waits for some method to notify it, while m' waits for the locks held by m[1]. In the literature, such a deadlock has also been called a *hold and wait* deadlock. We discuss two examples of such a deadlock.

[1]It is possible that there is a third method m'' in which `notify` is reachable, and it can issue a notification. However, while checking possibility for a deadlock between concurrent invocation of m and m', we err on the conservative side, and do not make assumptions about the existence of such a m''.

Example 8.1.2. Consider the code shown below. In Line 4, method `m1_w` releases the monitor `mon2`, but still holds `mon1`. As a result, `m1_n` can never reach Line 4 as it remains "stuck" in the entry set of `mon1` at Line 2. As both `m1_w` and `m1_n` cannot progress, this could lead to a deadlock.

```
1: public void m1_w () {
2:    synchronized (mon1) {
3:       synchronized(mon2) {
4:          mon2.wait();
5:       }
6:    }
7: }
```

```
1: public void m1_n () {
2:    synchronized (mon1) {
3:       synchronized (mon2) {
4:          mon2.notify();
5:       }
6:    }
7: }
```

Example 8.1.3. Methods `m2_w` and `m2_n` shown below have a similar situation as in Example 8.1.2. Method `m2_w` releases `mon1` in Line 4, but still holds `mon2`. Thus, `m2_n` cannot proceed beyond Line 3, where it gets stuck in the entry set of `mon2`.

```
1: public void m2_w () {
2:    synchronized (mon1) {
3:       synchronized(mon2) {
4:          mon1.wait();
5:       }
6:    }
7: }
```

```
1: public void m2_n () {
2:    synchronized (mon1) {
3:       synchronized (mon2) {
4:          mon1.notify();
5:       }
6:    }
7: }
```

Deadlock due to lock-order inversion. In addition to the deadlocks due to unreachable notification, nested monitors also cause deadlocks due to an inversion in the lock acquisition order. We explain this scenario in Example 8.1.4.

Example 8.1.4. Consider the following interleaved execution of the methods `m3_w` and `m3_n` shown below:

```
1: public void m3_w {
2:    synchronized (mon1) {
3:       synchronized(mon2) {
4:          mon1.wait();
5:       }
6:    }
7: }
```

```
1: public void m3_n {
2:    synchronized (mon1) {
3:       mon1.notify();
4:       synchronized(mon2) {
5:             ...
6:       }
7:    }
8: }
```

Method `m3_w` holds the monitor `mon2`, and is in the wait set of `mon1` at Line 4. Method `m3_n` acquires `mon1`, and then calls `mon1.notify()` (Line 3). Upon waking up, `m3_w` first tries to re-acquire `mon1`. However, as `m3_n` holds `mon1`, `m3_w` cannot proceed. On the other hand, `m3_n` tries to acquire `mon2`, but as `m3_w` holds `mon2`, `m3_n` cannot progress beyond Line 4. Effectively, due to the semantics of `wait`, the lock-acquisition order gets reversed, causing the classic cyclic dependency deadlock that we discussed in the previous sections.

In this section, we show how we can extend the lock-graph computation in Section 7.1 to capture deadlocks induced by nested monitors.

8.1.3 Extended Lock-Graph

We extend the notion of a lock-acquisition order graph defined in Section 7.1. Consider the following set of edges in $cfg(m)$:

$$e_l : \ (u_0 \xrightarrow{s_0 \ = \ \texttt{lock(mon)}} v_0)$$

$$e_w : \ (u_1 \xrightarrow{s_1 \ = \ \texttt{mon.wait()}} v_1)$$

$$e_n : \ (u_2 \xrightarrow{s_2 \ = \ \texttt{mon.notify()}} v_2)$$

We recall the rule for modeling the edge e_l in Rule R0 from Section 7.1. Recall that the set of locks held by m before executing a statement s is $\texttt{lockset}(u)$ for the edge $u \xrightarrow{s} v$ in $cfg(m)$. We denote the extended lock-graph by $elg(V, E)$. Rules R1,R2 specify how to model the edge e_w for a `wait` statement, while Rule R3 specifies how to model the edge e_n for a `notify` statement.

$$\forall \ell \in \texttt{lockset}(u_0), \qquad \text{add edge} \ \ (\ell, \texttt{mon}) \tag{R0}$$

$$\forall \ell \in \texttt{lockset}(u_1), \ \ell \neq \texttt{mon}, \quad \text{add edge} \ \ (\ell, \texttt{smon}) \tag{R1}$$

$$\forall \ell \in \texttt{lockset}(u_1), \ \ell \neq \texttt{mon}, \quad \text{add edge} \ \ (\ell, \texttt{mon}) \tag{R2}$$

$$\forall \ell \in \texttt{lockset}(u_2), \ \ell \neq \texttt{mon}, \quad \text{add edge} \ \ (\texttt{smon}, \ell) \tag{R3}$$

For a method m executing s_1, Rule R1 mimics the act of relinquishing `mon` and waiting. While m waits, it holds all the locks in $\texttt{lockset}(u_1)$ (except for `mon`). To model this, we create a vertex `smon` in the lock-graph, and add edges from every monitor ℓ (except `mon`) in $\texttt{lockset}(u_1)$ to `smon`. A method

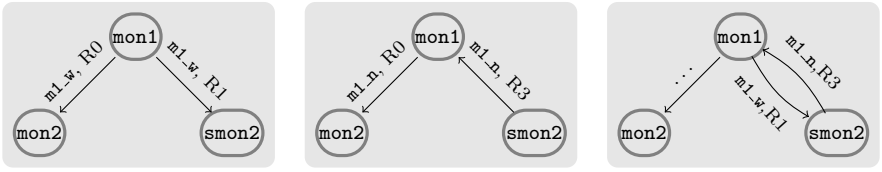
m' containing statement s_2 can reach s_2 if it can successfully acquire all the monitors (except `mon`) that *dominate* `mon.notify()`, *i.e.* all the monitors in `lockset`(u_2) except `mon`. We model this by adding edges from `smon` to each monitor in `lockset`(u_2) (Rule R3). Rules R1 and R3 ensure that if monitor ℓ is held by method m when it starts waiting, and if the method m' needs to acquire ℓ to reach statement `mon.notify()`, then $elg(m) \sqcup elg(m')$ contains a cycle of the form $\ell \rightarrow$ `smon` $\rightarrow \ell$.

Rule R2 encodes lock-order inversion. Suppose m holds certain monitors, and is waiting as a result of a call to `mon.wait()`. Upon waking up, m tries to re-acquire `mon`; so we add dependency edges $(\texttt{mon}', \texttt{mon})$, where `mon`$'$ is some monitor ($\neq$ `mon`) held by m before executing `mon.wait()` Recall from Section 7.2 that a cycle in the lock-graph indicates a potential deadlock. We show how a similar rule can be obtained for extended lock-graphs.

Def. 8.1.1 (Distinct Cycle)**.** A cycle in the merged extended lock-graph is called a distinct cycle if each edge in the cycle is induced by a distinct method (invocation).

Lemma 8.1.1 (Generalized Nested Monitor Rule)**.** *Let the extended lock-graphs obtained using Rules R1, R2, R3 for methods $m_1, \ldots, m_k$ be denoted by $elg(m_1), \ldots, elg(m_k)$. Concurrent calls to methods $m_1, \ldots, m_k$ may deadlock if $\bigsqcup_{i=1}^{k} elg(m_i)$ contains a distinct cycle.*

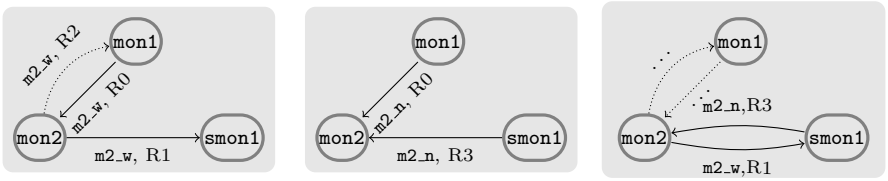
Example 8.1.5. Consider the extended lock-graphs for `m1_w` and `m1_n` from Example 8.1.2:



$lg(\texttt{m1\_w})$ $lg(\texttt{m1\_n})$ $lg(\texttt{m1\_w}) \sqcup lg(\texttt{m1\_n})$

We can observe that there is a distinct cycle in $\text{lg}(\texttt{m1\_w}) \sqcup lg(\texttt{m1\_n})$.

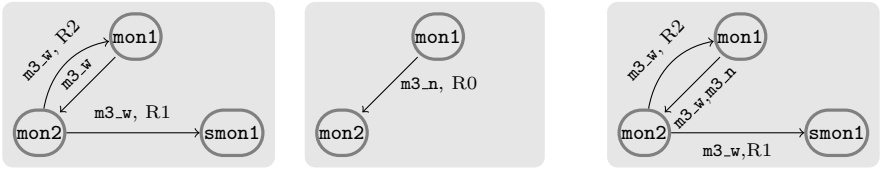
Example 8.1.6. Consider the extended lock-graphs for `m2_w` and `m2_n` from Example 8.1.3:



$lg(\texttt{m2\_w})$ $lg(\texttt{m2\_n})$ $lg(\texttt{m2\_w}) \sqcup lg(\texttt{m2\_n})$

In this case, there are two distinct cycles in the merged graph indicating potential deadlocks. The cycle due to unreachable notification corresponds to the solid edges.

Example 8.1.7. Consider the extended lock-graphs for `m3_w` and `m3_n` from Example 8.1.4:



$lg(\texttt{m3\_w})$ $lg(\texttt{m3\_n})$ $lg(\texttt{m3\_w}) \sqcup lg(\texttt{m3\_n})$

There is a distinct cycle between nodes mon1 and mon2 by picking the label m3_n for the edge (mon1, mon2) and m3_w for (mon2, mon1), which indicates a potential deadlock.

8.2 Extended Lock-Graph Extraction and Encoding

In this section, we first discuss the modifications to the static analysis algorithm in order to extract extended lock-graphs from library methods. We follow this by a discussion on the modifications to the symbolic encoding scheme outlined in Section 7.2.1 to accommodate extended lock-graphs.

8.2.1 Modifications to Lock-Graph Computation

The algorithm for lock-graph computation outlined in Section 7.1 can be extended to accommodate the generalized nested monitor rule. Recall that the summary computed for each point in the control-flow graph of a method m is a tuple $(lg(V, E), \texttt{lockset}, \texttt{roots})$ where $lg(V, E)$ is the lock-order graph, lockset is the set of locks currently acquired by method m, and roots is the set of locks that do not have parent nodes in lg. For computing the extended lock-graphs, the partial summary is defined as the tuple: $(elg(V, E), \texttt{lockset}, \texttt{roots}, \texttt{notifySet})$, where elg is the extended lock-graph as defined in Section 8.1.3, and notifySet is the set of monitors on which mon.notify() has been invoked.

The extendedComputeFlow function specifies the flow equations for the wait-notify statements, and modified flow equations for method calls. For a

Function `extendedComputeFlow`

input : *cfg* edge: $u \xrightarrow{s} v$, Method name: m
output : partial summary *out*

```
1  begin
2      in :=psum(u), out :=∅
       /* in = (V_i, E_i, roots_i, lockset_i, notifySet_i), out =
          (V_o, E_o, roots_o, lockset_o, notifySet_o) */
3      switch (s) do
4          Lines 7-19 of function computeFlow
           /* Wait Statement.  */
5          case mon.wait() ():
6              foreach (mon' ∈ lockset_i) s.t. (mon' ≠ mon) do
                   /* Rule R1:  */
7                  E_o :=E_o ∪ {(mon' --m--> smon)}
                   /* Rule R2:  */
8                  E_o :=E_o ∪ {(mon' --m--> mon)}
           /* Notify Statement.  */
9          case mon.notify() ():
10             foreach (mon' ∈ ls_i) s.t. (mon' ≠ mon) do
                   /* Rule R3:  */
11                 E_o :=E_o ∪ {(smon --m--> mon')}
12                 notifySet :=notifySet ∪ {smon}
           /* Method Invocation.  */
13         case m'(a_1, ..., a_k) :
14             Lines 13-19 of function computeFlow
               /* Recall, sum' = summary(m')|_{∀i:f_i↦a_i}  */
               /* Also, sum' = (V', E', lockset', roots', notifySet')  */
15             foreach smon ∈ notifySet' do
16                 foreach mon ∈ lockset do
17                     E_o :=E_o ∪ {(smon --m--> mon)}
18             notifySet :=notifySet ∪ notifySet'
19 end
```

given method m_1, the edges induced by Rules R1 and R2 are fully contained within $\texttt{summary}(m_1)$, and are added in standard fashion (Lines 7, 8). However, the edges induced by Rule R3 are "reverse" edges that can point to nodes outside of $\texttt{summary}(m_1)$. Consider the case where m calls m_1 and m_1 contains a `mon.notify()` statement. Now, by Rule R3, we add edges from `mon` (which is a node in $elg(m_1)$) to every lock in `lockset` at the call-site of m_1 (which is in m). To ease this computation, we add `mon` to the set `notifySet`, when `mon.notify()` is called (Line 12). Let $u \xrightarrow{s=m_1(a_1,\ldots,a_k)} v$ be the call-site of method m_1 in $cfg(m)$. When we concatenate $\texttt{summary}(m_1)$ with the partial summary at point u in m (*i.e.* $\texttt{psum}(u)$), we add edges from the set `notifySet` in $\texttt{summary}(m_1)$ to every monitor in $\texttt{lockset}(u)$ in $\texttt{psum}(u)$ (Line 17-17).

8.2.2 Symbolic Encoding Revisited

Recall that for a given pair of methods m_1 and m_2, there is a potential deadlock if $elg(m_1) \sqcup elg(m_2)$ contains a *distinct cycle*. For deadlockability analysis, we can extend the above rule as follows: α is a deadlock-causing aliasing pattern for methods m_1 and m_2 if $\alpha \triangleright elg(m_1) \sqcup elg(m_2)$ contains a distinct cycle. We can extend the symbolic encoding described in Section 7.2.1 to encode the extended lock-graphs. The main difficulty is that an extended lock-graph $elg(m)$ for a method m can have cycles, while our symbolic encoding for graphs and aliasing patterns relies on encoding *acyclic* graphs.

We observe that a cycle is added to $elg(m)$ due to simultaneous application of Rule R0 and R2. For instance, consider the case where m acquires

monitor mon1 followed by mon2, and then executes mon1.wait(). Rule R0 requires an edge to be added from mon1 to mon2, while Rule R2 requires an edge to be added from mon2 to mon1. To check for existence of a distinct cycle, we need either the edge $(\texttt{mon1}, \texttt{mon2})$, *or*, the edge $(\texttt{mon2}, \texttt{mon1})$, but not both. In effect, we can decompose $elg(m)$ into two acyclic graphs $elg_1(m)$ and $elg_2(m)$, each of which contains exactly one of these two edges. It is easy to see that such a disjunctive decomposition of the extended lock-graph can be systematically performed by "breaking cycles" formed by the symmetric edges induced by R0 and R2.

Example 8.2.1. Consider the method m2_w shown in Example 8.1.3, and its corresponding extended lock-graph shown in Example 8.1.6. The encoding for lg(m2_w) is expressed as two conjunctions as follows:

$$\Psi(G_1) = \underbrace{x(\texttt{mon1}) < x(\texttt{mon2})}_{R0} \ \wedge \ x(\texttt{mon2}) < x(\texttt{smon1})$$
$$\Psi(G_2) = \underbrace{x(\texttt{mon2}) < x(\texttt{mon1})}_{R2} \ \wedge \ x(\texttt{mon2}) < x(\texttt{smon1})$$

Example 8.2.2. Continuing with Example 8.1.6 and Example 8.2.1, let α be the empty aliasing pattern. Consider the merged lock-graphs obtained by merging G_1 and G_2 individually with the lock-graph for m2_n.

$$\Psi(\alpha, G_1 \sqcup lg(\texttt{m2\_n})) = \left[\begin{array}{lcl} x(\texttt{mon1}) < x(\texttt{mon2}) & \wedge & x(\texttt{mon2}) < x(\texttt{smon1}) \\ x(\texttt{mon1}) < x(\texttt{mon2}) & \wedge & x(\texttt{smon1}) < x(\texttt{mon2}) \end{array} \right]$$
$$\Psi(\alpha, G_2 \sqcup lg(\texttt{m2\_n})) = \left[\begin{array}{lcl} x(\texttt{mon2}) < x(\texttt{mon1}) & \wedge & x(\texttt{mon2}) < x(\texttt{smon1}) \\ x(\texttt{mon1}) < x(\texttt{mon2}) & \wedge & x(\texttt{smon1}) < x(\texttt{mon2}) \end{array} \right]$$

We can see that both conjunctions are unsatisfiable. Moreover, each conjunction encodes a distinct cycle in the merged extended lock-graph.

Example 8.2.3. Consider Example 8.1.7. Let α be the empty aliasing pattern. Note that $lg(\texttt{m3\_w})$ in Example 8.1.7 is identical to $lg(\texttt{m2\_w})$ in Example 8.1.6, and thus the decomposition of $elg(\texttt{m3\_w})$ into graphs G_1 and G_2 is as in Example 8.2.1. Now consider the merged lock-graphs obtained by merging G_1 and G_2 with $lg(\texttt{m3\_n})$ from Example 8.1.7.

$$\Psi(\alpha, G_1 \sqcup lg(\texttt{m3\_n})) = \left[\begin{array}{lcl} x(\texttt{mon1}) < x(\texttt{mon2}) & \wedge & x(\texttt{mon2}) < x(\texttt{smon1}) \ \wedge \\ x(\texttt{mon1}) < x(\texttt{mon2}) & & \end{array} \right]$$
$$\Psi(\alpha, G_2 \sqcup lg(\texttt{m3\_n})) = \left[\begin{array}{lcl} x(\texttt{mon2}) < x(\texttt{mon1}) & \wedge & x(\texttt{mon2}) < x(\texttt{smon1}) \ \wedge \\ x(\texttt{mon1}) < x(\texttt{mon2}) & & \end{array} \right]$$

We can see that the second conjunction is unsatisfiable, and corresponds to a distinct cycle in the merged extended lock-graph for t_1 and t_2.

Thus, we can observe that if the extended lock-graph $elg(m_1)$ for a method m_1 has cycles, then we can decompose it into components $\{elg_1(m_1)$, ..., $elg_n(m_1)\}$, s.t. each $elg_i(m_1)$ is acyclic and $elg(m_1)$ is the union of the components, *i.e.*, $elg(m_1) = \bigcup_{i=1}^{n} elg_i(m_1)$. We can see from the above examples that for a pair of methods m_1, m_2, the merged extended lock-graph

$elg(m_1) \sqcup elg(m_2)$ has a distinct cycle if there exist some acyclic components $elg_i(m_1)$ and $elg_j(m_2)$ such that $elg_i(m_1) \sqcup elg_j(m_2)$ has a cycle. This is formalized in the theorem below.

Theorem 8.2.1. *Let G_i be some acyclic component of $elg(m_1)$ and G_j be some acyclic component of $elg(m_2)$. Let G_{ij} denote $G_i \sqcup G_j$. The formula $[\Psi(\alpha, G_{ij})]$ is satisfiable for all i, j iff $[\alpha \triangleright (elg(m_1) \sqcup elg(m_2))]$ does not have a distinct cycle.*

Proof. We give a proof outline:

1. We first prove that $[\alpha \triangleright elg(m_1) \sqcup elg(m_2)]$ does not contain a distinct cycle iff $\forall i$ and $\forall j$, $elg_i(m_1) \sqcup elg_j(m_2)$ does not contain a cycle. This follows from the definition of the decomposition operation.

2. If $elg_i(m_1) \sqcup elg_j(m_2)$ does not contain a cycle, then by Theorem 7.2.1, we know that $\Psi(\alpha, G_{ij})$ is satisfiable. Thus if $\forall i$ and $\forall j$, if $elg_i(m_1) \sqcup elg_j(m_2)$ does not contain a cycle, then $\forall i$ and $\forall j$, $\Psi(\alpha, G_{ij})$ is satisfiable.

□

8.3 Bibliographic Notes

In a short paper raising some of the issues with nested monitors [103], the author points out that there are potential deadlocks due to nested monitors and discusses one such situation. Some of the issues raised by this paper are addressed in [77]. While the author convinces the reader that nested monitors

are indeed an important structuring device for concurrent methods, the paper admits the need for further analysis to elucidate the rules required for avoiding deadlocks.

[15] presents a discussion on structuring modular concurrent programs, and proposes that a shared resource abstraction should contain the implementation of the synchronization scheme, as well as the definitions of the internal structure and operations of the resource. It further argues that monitors only satisfy this structure partially, and that if shared resources were to be implemented as suggested, nested monitor deadlocks would be greatly reduced. Though this idea seems promising, it does not seem to have become popular in concurrent programming languages such as `Java` and `C/pthreads`.

[22] presents a type system for deadlock-free execution of programs. The type system requires programmers to partition locks into a fixed number of equivalence classes and specify a partial order between the classes, while the type checker statically verifies if these conditions are met. The approach also allows for lock classes to be specified in a tree order, which can be dynamically updated at run-time. While such a method could be very useful for developing deadlock-free applications in the future, it is not clear if it could be applied to analyzing existing concurrent software. Furthermore, it could be argued that specifying such partial orders and lock equivalence classes could be difficult for an inexperienced developer.

[33] shows the application of the Bandera tool-set to model check `Java` programs with the help of property-directed slicing algorithms. [144] discusses

`Java PathFinder` (JPF) which intergrates program analyses, model checking and testing. It can handle real `Java` programs, and integrates deadlock detection as a part of the analysis algorithm. The paper says that JPF can handle programs that are about 1000 to 5000 lines in range. For a model checking-based exhaustive verification tool, this is quite impressive. However, we have used static analysis-based (conservative) techniques on libraries with millions of lines of `Java` code, and it seems unlikely to obtain the coverage and precision of model checking for libraries of this size with the current state of the art.

Finally, [148], which presents deadlockability analysis with type-based abstractions, contains a rule to model `wait-notify` statements. This rule essentially captures the lock-order inversion due to nested monitors in our approach. The authors also track `wait` statements that may occur outside the scope of a monitor. Such `wait` statements result in a run-time exception in `Java`. We could easily add similar static rules to detect such ill-formed synchronization patterns that pose a threat to thread safety in our approach; however, this is orthogonal to the issue of deadlockability, and thus not included in our current algorithms.

Chapter 9

Experimental Evaluation

We have implemented a prototype tool for synthesizing interface contracts for `Java` libraries. The tool consists of a summary based lock-order graph analysis using summarization, as outlined in Section 7.1. The lock-graphs are then encoded into logical formulae for symbolic enumeration of aliasing patterns, as outlined in Section 7.2.

The tool is designed as a plug-in into the `soot` framework for implementing the lock-order graph extraction [140]. Must-aliases for lock objects are tracked across methods using our own analyzer built using the `cg.spark` intraprocedural alias analysis phase within `soot`. We augment `spark` to track aliasing between fields, yielding a field-sensitive analysis. Before generating constraints for analysis with the SMT solver, we prune the lock-order graphs using various filtering strategies strategies (in addition to those discussed in Section 7.3):

(a) Pruning *unaliasable* fields (*e.g.*, `final` fields).

(b) Removing objects declared `private` that are not accessed outside the constructor or finalizer.

(c) Removing immutable string constants and `java.lang.Class` constants.

(d) Pruning objects that cannot escape the scope of a given library method using an *escape analysis*.

These filtering strategies are sound: our tool does not miss any potential deadlock due to these strategies. The generated constraints are solved using the SMT solver `Yices` [47]. Table 9.1 summarizes the potential deadlocks thus obtained. Table 9.1 shows that our tool runs in a relatively short amount of time even for large `Java` libraries; we could analyze about over a *million lines of code* in about 48 minutes. Furthermore, the runtime is dominated by the lock-order graph computation rather than the enumeration and constraint solving with the SMT solver. Thus, the most time-consuming phase in deadlockability analysis, *i.e.*, explicit reasoning over lock-graphs, is made highly efficient with the help of our symbolic reasoning techniques.

Some deadlock-causing aliasing patterns are *false positives*. These patterns result from the following main sources: a) the static lock-order graph construction is a *may* analysis, and hence there are inaccurate edges and nodes in the lock-order graph, b) our alias analysis is a *may* analysis, which leads to aliasing patterns that cannot be realized, and c) there could be *gated cycles*, *i.e.*, cycles having a common lock as a prefix, which prevents them from being simultaneously reachable. We manually examine the output of our tool to discard such patterns. However, the output of our tool may also consist of a large number of "redundant" deadlock-causing patterns. These patterns

Table 9.1: Experimental Results

Library	KLOC	Num. of Aliasing Patterns		Time Taken (secs)[a]		Unique Scenarios	
		Checked	Deadlock-causing	Lock-Graph	SMT	False Positives	Potential Deadlocks
apache-log4j	33.3	4	4	130	0.1	1	**1**
cache4j	2.6	0	0	15	-	-	**-**
ftpproxy	1.0	0	0	13	-	-	**-**
hsqldb	157.6	369	231	804	2.8	3	**3**
JavaFTP	2.6	0	0	9	-	-	**-**
netty	11.0	0	0	14	-	-	**-**
oddjob	41.3	0	0	250	-	-	**-**
java.applet	0.9	102	64	64	1.0	1	**1**
java.awt	163.9	5325	3800	454	26.4	2	**3**
java.beans	16.2	148	108	31	1.5	1	**2**
java.io	28.6	32	0	39	0.0	-	**-**
java.lang	55.0	279	89	46	1.9	3	**2**
java.math	9.1	0	0	18	-	-	**-**
java.net	26.5	55	44	32	0.5	1	**1**
java.nio	46.7	0	0	19	-	-	**-**
java.rmi	9.1	2	2	14	0.1	1	**0**
java.security	34.2	0	0	27	-	-	**-**
java.sql	22.2	1836	0	10	8.0	-	**-**
java.text	22.6	26	18	26	0.2	1	**0**
java.util	116.8	188	117	190	2.0	4	**3**
javax.imageio	24.7	0	0	22	-	-	**-**
javax.lang	5.2	0	0	8	-	-	**-**
javax.management	67.5	16	6	74	0.2	2	**0**
javax.naming	19.5	0	0	64	-	-	**-**
javax.print	2.1	2	0	27	-	-	**-**
javax.security	11.7	164	110	27	1.2	2	**0**
javax.sound	14.3	0	0	10	-	-	**-**
javax.sql	18.2	0	0	14	-	-	**-**
javax.swing	322.2	132	120	353	1.6	2	**2**
javax.xml	48.9	0	0	27	-	-	**-**

[a]All experiments were performed on a Linux machine with an AMD Athlon 64x2 2.2 GHz processor, and 6GB RAM.

that are repeated instantiations of the same underlying deadlock scenario, and appear due to the fact that several library methods typically invoke the same deadlock-prone utility method. Such a deadlock gets reported multiple times in our current implementation, each under a different set of library entry methods. The table shows the number of unique scenarios after considering such redundancies. While the process of identifying unique scenarios can be automated; presently, we identify such scenarios by manual inspection.

Example 9.0.1. From the lock-order graph of `postEventPrivate` presented in Figure 6.3, if we concurrently invoke the method `postEventPrivate` on two separate objects `a` and `b`, then it leads to a deadlock under a specific aliasing pattern. However, the methods `postEvent`, `push` and `pop` in the same class also invoke the `postEventPrivate` method, and hence are susceptible to the *same* deadlock. Thus, for each pair of these methods, the same underlying deadlock-causing aliasing pattern is generated. In our experiments, we observed 324 possible deadlock-causing aliasing patterns, all of which correspond to this single unique scenario involving calls to `postEventPrivate`.

Our tool predicts deadlocks that are highly relevant to some of the clients using the libraries that we have analyzed. Some have already manifested in real client code, and have been reported as bugs by developers in various bug repositories. Table 9.2 summarizes the library name and the bug report locations we have found using an internet search. Inspection of the bug reports reveals that the aliasing patterns at the call-sites of the methods involved in the

Library Name	Method names	Bug Report
java.awt (EventQueue)	postEventPrivate, wakeup	[139]:4913324 [139]:6424157, [139]:6542185
java.awt (Container)	removeAll, addPropertyChangeListener	[117]
java.util (LogManager) (Logger)	addLogger getLogger	[139]:6487638
javax.swing (JComponent)	setFont paintChildren	Jajuk [89]
hsqldb (Session)	isAutoCommit close	[116]

Table 9.2: Real Client Deadlocks

deadlock, correspond to a violation of the interface contract for that library, as generated by our tool.

Part IV

Conclusions

Chapter 10

Conclusions

In this chapter, we conclude with a summary of the important results, and indicate open problems and future directions of work.

10.1 Summary of Results

This dissertation presents techniques for tractable verification of software. In contrast to hardware verification, which deals with an essentially finite-state (albeit computationally intensive) problem, software verification is usually undecidable. The sources of undecidability are typically heap-allocated data structures, arbitrary recursion, and concurrency. We focus on particular problems within the "usual suspects" for undecidability, and give directed solutions for efficient and automatic verification for fragments that cover a large range of useful sequential and concurrent software libraries.

The dissertation is broadly divided into two parts: (1) verification of pre/post-condition based specifications for methods in sequential data structure libraries, and (2) verification of thread safety in concurrent libraries through an investigation into techniques for deadlock analysis. We now summarize the key results for each part.

10.1.1 Sequential Verification: Contributions

I. **Automata-theoretic Framework:**

We formulate a general automata-theoretic framework for the verification of methods acting on sequential data structures. The solution strategy can be described in the following high-level steps:

(a) data structures are modeled as directed, labeled graphs,

(b) specifications are provided as a pre-condition automaton $\mathcal{A}_\varphi$ specifying valid input graphs, and a post-condition automaton $\mathcal{A}_{\neg\psi}$ specifying *invalid* output graphs,

(c) a method $\mathcal{M}$ is represented as a *method automaton* $\mathcal{A}_\mathcal{M}$ that accepts a composite graph $G_c = (G_i, G_o)$ iff $G_o = \mathcal{M}(G_i)$,

(d) a product automaton $\mathcal{A}_p = \mathcal{A}_\varphi \otimes \mathcal{A}_\mathcal{M} \otimes \mathit{Anpost}$ is constructed; $\mathcal{A}_p$ accepts a composite graph $G_c = (G_i, G_o)$ iff G_i satisfies the pre-condition, G_o is obtained by the action of $\mathcal{M}$ on G_i, and G_o *fails to* satisfy the post-condition,

(e) the product automaton is tested for emptiness; if $\mathcal{A}_p$ is empty, then the method is *correct*, otherwise, a *counterexample* to the correct operation of $\mathcal{M}$ is obtained from $\mathcal{A}_p$.

II. **Undecidability Result:**

The automata-theoretic framework as outlined above, in theory, works for any arbitrary method, and for arbitrary specifications. However, the method automata and the specification automata in themselves, and when combined in

the product construction, may not have decidable emptiness problems. For instance, the exact automaton to model a certain method may be a *linear bounded automaton*, or a 2-*headed automaton*. Both these automata have undecidable emptiness problems. As another example, if the method automaton is a finite state automaton, but if both the pre-condition and post-condition automata are pushdown automata, checking the emptiness of the product is undecidable. Thus, we show that the general verification problem for methods operating on parameterized data structures is *undecidable*.

III. **Specifications:**

In our technique specifications can be data structure invariants or more nuanced pre/post-conditions that specify valid input graphs and corresponding valid output graphs. While our preferred formalism for specifications is that of tree automata and its variants, we also allow the use of any logic-based formalism (such as temporal logics CTL and μ-calculus), which can be translated into tree automata at a small cost. Example specifications include: shape properties such as acyclicity, tree-ness, sorted-ness, list-ness, data-centric properties such as: "no red node has a red child", "there is no node with the data value '`a`' in the data structure", reachability of data values and pointer values, and memory-related properties such as absence of dangling pointers, null pointer dereferences, and (some types of) memory leaks.

IV. **Methods:**

We define the scope of our technique in terms of mathematical conditions on

methods allowed (details are included in the next two contributions). Our framework allows us to verify methods that include insertion, deletion, data-value modification of nodes in a vast array of data structures including linked lists, binary search trees, general trees, balanced trees, directed acyclic graphs, and general directed graphs. We can also verify methods to reverse linked lists and rotate balanced trees.

V. **Decidable Fragments for Iterative Methods:**
We formulate mathematical conditions on methods that, if fulfilled, ensure efficient and automatic verification. In particular, we postulate that if (a) methods perform only a bounded number of destructive updates on the underlying data structure, (b) use only localized updates, and (c) terminate, then the methods can be mimicked by finite state tree automata. Of these, we show that it is undecidable to determine if (a) is true, we provide a programming language (Bud-Pl) to syntactically enforce (b) and assume that a proof for (c) is available through an oracle. The last assumption is typically not an obstacle as there are techniques to prove termination of methods on data structures [75].

VI. **Decidable Fragments for Recursive Methods:**
We provide a syntactic fragment for recursive methods operating on tree-like data structures. We guarantee that a method in our fragment can always be verified automatically, as our fragment ensures that the method performs only a bounded number of destructive updates on the underlying data structure.

Since this fragment subsumes tail-recursive methods (which in turn can be used to model iterative methods), this provides us with a decidable syntactic fragment for verifying iterative methods on tree-like data structures.

VII. **Compilation Algorithms:**

We provide algorithms to mechanically compile a method specified in our syntax into method automata. Separate algorithms are provided for each of the fragments outlined above, and the time complexity of each algorithm is linear in the size of the method.

VIII. **Experimental Validation:**

We give the results of experiments performed with a prototype tool (called Pravda) that is capable of verifying useful iterative and recursive methods against pre/post-condition based specifications.

10.1.2 Concurrent Verification: Contributions

I. **Deadlockability Analysis:**

We formally define deadlockability analysis for concurrent libraries. The purpose of this analysis is to identify if, for *any* client program using a concurrent library, there are calling contexts for library methods which leads to a library-level deadlock. A form of deadlockability analysis was introduced in [148]. This work uses *types* of syntactic expressions corresponding to object monitors as conservative approximations for the alias information. Though the authors are able to identify important potential deadlocks, their approach is

susceptible to false positives, which have to be then filtered using (possibly unsound) heuristics.

II. **Interface Contracts:**

The goal of the deadlockability analysis that we perform is to derive *interface contracts*. These are logical expressions that encode aliasing patterns between the arguments of a set of methods that can lead to deadlocks during concurrent invocation of these methods. These contracts can be used variously: (a) statically checked in a client code to detect possible deadlocks in the client application, (b) dynamically enforced at run-time in a client, to prevent deadlocks, (c) as documentation for the developers of client code that use the library.

III. NP**-completeness of Aliasing Pattern Enumeration:**

Static deadlock analysis typically involves obtaining lock-order graphs for methods, and inspecting the merged lock-order graph for cycles. Existence of cycles indicates a possible deadlock during the concurrent invocation of the methods of interest. Analyzing methods in a library for possible deadlock-causing aliasing patterns between the method arguments (for a pair of methods) can thus be formulated as: (a) initialize the set of unexamined aliasing patterns to all aliasing patterns between the pair of lock-order graphs, (b) for each unexamined aliasing pattern check if the merged lock-graph obtained fusing the aliased nodes contains a cycle, (c) halt if there are no unexamined aliasing patterns. We show that the problem of enumerating aliasing

patterns (effectively checking if there is any unexamined aliasing pattern) is NP-complete.

IV. **Efficient Enumeration through Symbolic Reasoning:**
We show how we can encode aliasing patterns and lock-order graphs into a theory of integers (UTVPI), and reduce cycle detection in a lock-graph to a SAT problem amenable to Satisfiability Modulo Theory (SMT) solvers. Further, we show how we can reduce the set of aliasing patterns to be considered by a notion of subsumption, and a host of sound pruning techniques to reduce the sizes of the lock-graphs. Finally, we derive interface contracts from the set of deadlock-causing aliasing patterns.

V. **Deadlocks in Signaling-based Synchronization:**
We formulate syntactic rules for predicting deadlocks in methods that use `wait-notify` statements for signaling-based cooperative synchronization. We term the set of static rules that we derive, the *generalized nested monitors rule*. We give the precise static analysis for deriving extended lock-graphs that encode the dependencies between locks and `wait-notify` as per the above rule. We present a symbolic encoding for extended lock-graphs that enables efficient enumeration of deadlock-causing aliasing patterns and consequently the interface contracts for methods that use `wait-notify` statements.

VI. **Experimental Validation:**
We present experimental results obtained with a deadlockability analysis tool for `Java` libraries. Our tool demonstrates that our approach is more accurate

than previous approaches, and at the same time has remarkable performance on real-world concurrent libraries. Our tool is able to synthesize interface contracts for deadlock-free execution with a low number of false positives for over a million lines of `Java` code in less than 50 minutes. Significantly, our tool predicts deadlocks that have already manifested in real client code, and have been reported as bugs by developers in various bug repositories. Inspection of the bug reports reveals that the aliasing patterns at the call-sites of the methods involved in the deadlock, correspond to a violation of the interface contract for that library, as generated by our tool.

10.2 Open Problems and Future Work

There are a number of interesting open problems that can be thought of as a ready extension of the techniques presented in this dissertation. As before, we divide them into two parts:

10.2.1 Data-Structure Manipulating Methods

1. In the case of iterative methods, our approach assumes the existence of an oracle that determines if the method performs a bounded number of destructive updates, or if it terminates. We derive a syntactic fragment that guarantees the bounded updates property for tree-like data structures. Whether such a syntactic fragment exists for a more general class of graphs is an open problem. Combinations of our technique with techniques that detect termination for heap-manipulating programs is

another area that needs further exploration.

2. Verification of recursive methods on directed acyclic graphs (*dags*) is hard, as *dags* can contain sharing between nodes. The restriction of "one visit per successor" is not enough to ensure the bounded updates property in *dags*. However, if we can enforce (or guarantee) that a method visits each node in a *dag*, and thus every sub-*dag*, at most once, then such methods would satisfy the bounded updates property. These could be then verified using a modified form of the algorithms presented here. Such a restriction is common for methods manipulating *dags*, and is usually implemented with the help of a marking scheme employing Boolean-valued `visited` fields in the data structure nodes. The key to automatic verification of recursive methods on *dags* thus lies in the single visit property; however, the following problems are open: (a) Is there an elegant syntactic class of methods (that does not not use `visited` fields) with the single visit property? (b) Is there a translation from methods that use `visited` fields to finite-state method automata? For (b), note that since an automaton mimicking such a method would have to remember every node that has been marked *visited*, a simplistic finite-state formulation would not work.

3. Most interesting properties of data structures can be specified using non-deterministic tree automata on finite and infinite trees. However, certain properties such as: "Is the tree balanced?" are finite state tree automata

ineffable [50]. For such properties, a class of automata called the *deterministic bottom-up tree automata with constraints between brothers* (AWCBB) [31, 16] looks promising. These automata are bottom-up tree automata with a polynomial time emptiness algorithm, and are capable of accepting full balanced trees. Alternatively, tree automata with size constraints [76] also have good closure properties (under intersection); however, the decision procedure for emptiness of these automata has high complexity.

4. Pushdown tree-walking automata [141] are an interesting class of automata that can be used to model certain methods. As opposed to conventional tree automata that split into multiple copies at each child node, these automata have only one copy, that *walks* along the tree. It also has a pushdown, with the restriction that the moves of the pushdown are synchronized with the moves of the automaton, *i.e.*, symbols are pushed when the automaton visits a child node, and popped when the automaton returns back to the parent. These automata seem like a more natural model of recursive methods on trees (and conceivably directed acyclic graphs), and could be used to relax some of the restrictions we place (such as disallowing updates to parent nodes). These automata are exponentially more succinct than conventional tree automata; thus, they do incur an exponential increase in cost in the nonemptiness procedure.

5. In our current framework, we require each of $\mathcal{A}_{\mathcal{M}}$, $\mathcal{A}_{\varphi}$ and $\mathcal{A}_{\neg\psi}$ to be finite state (tree) automata. We can relax this requirement so that any

one of these is a pushdown (tree) automaton. Thus, for instance, we could have a richer class of methods, that can be modeled by pushdown automata (in the case of lists) or pushdown tree automata (in the case of general graphs). In the former case, the complexity of our technique is still polynomial [85], while in the latter, the complexity is exponential [120].

6. It is known that the deterministic $o(log\ log\ n)$ space complexity class equals the class of regular languages. In this dissertation, we show that if a method modifies the underlying data structure a bounded (by a constant) number of times, then effectively the language of the updates by the method is regular. In other words, we exploit that the class of regular languages $\textsc{Reg} = \textsc{Dspace}(O(1))$. However, $\textsc{Reg} = \textsc{Dspace}(o(log\ log\ n))$, which is a looser upper bound. If we can identify methods that are contained within this complexity class, we can still mimic them by finite state automata. It would be interesting to identify examples of such methods and possibly obtain syntactic fragments for such methods.

7. Finally, from an engineering perspective, it would be worthwhile to be able to broaden the scope of our framework to verify methods written in `C`, `C++` or `Java`. Techniques such as bottom-up shape analysis [73] could be used to summarize portions of code that are not compliant with our syntactic restrictions. Thus, the overall framework could leverage the

power of exact verification offered by our framework to programs that do not satisfy our stipulations.

10.2.2 Concurrency Verification

1. Synchronization primitives such as locks and monitors used for enforcing mutual exclusion are the most common source of deadlocks. Hence, the main thrust of this dissertation is on detecting deadlocks based on cyclic dependencies in the acquisitions of such (lock and monitor) variables. Extension to other synchronization primitives such as (counting) semaphores, barriers, locks with different re-entrancy models is an open problem.

2. While the number of false positives generated by our tool is low, cases such as guarded cycles, *i.e.*, cycles that are infeasible as each entry node in the cycle is protected by a common lock [10], are not currently handled. While it is possible to examine lock-order graphs and aliasing patterns automatically to rule out guarded cycles, it would be neater to encode this as an additional constraint to the SMT solver. The exact formulation remains open.

3. Dealing with newer features of the `Java` language such as *generics*, and `Java`'s concurrency library (`java.util.concurrent`) that uses constructs similar to the `pthread` library is a challenge for future work. Automatic identification of unique scenarios from the interface contracts generated

by our current implementation would further reduce the manual effort required to process the output of our tool.

4. Empirical validation of the static analysis that checks/enforces the derived interface contracts on real client code would be an important part of future work.

Bibliography

[1] P. Abdulla, A. Bouajjani, J. Cederberg, F. Haziza, and A. Rezine, "Monotonic Abstraction for Programs with Dynamic Memory Heaps," in *Proc. of Computer Aided Verification (CAV)*, 2008, pp. 341–354.

[2] R. Agarwal and S. D. Stoller, "Run-time Detection of Potential Deadlocks for Programs with Locks, Semaphores, and Condition Variables," in *Proc. of Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD)*, 2006, pp. 51–60.

[3] R. Agarwal, L. Wang, and S. D. Stoller, "Detecting Potential Deadlocks with Static Analysis and Run-Time Monitoring," *Hardware and Software, Verification and Testing*, pp. 191–207, 2006.

[4] R. Alur and P. Madhusudan, "Visibly Pushdown Languages," in *Proc. of Symposium on Theory of Computing (STOC)*, 2004, pp. 202–211.

[5] C. Artho and A. Biere, "Applying Static Analysis to Large-Scale, Multi-Threaded Java Programs," in *Proc. of the 13th Australian Conference on Software Engineering*, 2001, p. 68.

[6] I. Balaban, A. Pnueli, and L. Zuck, "Shape Analysis by Predicate Abstraction," in *Proc. of Verfication Model Checking and Abstract Interpretation (VMCAI)*, 2005, pp. 164–180.

[7] ——, "Shape Analysis of Single-Parent Heaps," in *Proc. of Verfication Model Checking and Abstract Interpretation (VMCAI)*, 2007, pp. 91–105.

[8] T. Ball and S. Rajamani, "Automatically validating temporal safety properties of interfaces," in *Proc. of* SPIN *Workshop on Model Checking of Software (SPIN)*, 2001, pp. 103–122.

[9] M. Benedikt, T. Reps, and M. Sagiv, "A Decidable Logic for Describing Linked Data Structures," in *Proc. of European Symposium on Programming (ESOP)*, 1999, pp. 2–19.

[10] S. Bensalem and K. Havelund, "Dynamic Deadlock Analysis of Multi-Threaded Programs," in *Proc. of the Haifa Verification Conference*, 2005, pp. 208–223.

[11] J. Berdine, C. Calcagno, and P. W. O'Hearn, "A Decidable Fragment of Separation Logic," in *Proc. of Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, 2004, pp. 97–109.

[12] O. Bernholtz and O. Grumberg, "Branching Time Temporal Logic and Amorphous Tree Automata," in *Proc. of the International Conference on Concurrency Theory (CONCUR)*. Springer, 1993, pp. 262–277.

[13] D. Beyer, T. Henzinger, R. Jhala, and R. Majumdar, "The Software Model Checker Blast," *Software Tools for Technology Transfer (STTT)*, vol. 9, no. 5, pp. 505–525, 2007.

[14] J. Bingham and Z. Rakamaric, "A Logic and Decision Procedure for Predicate Abstraction of Heap-Manipulating Programs," in *Proc. of Verfication Model Checking and Abstract Interpretation (VMCAI)*, 2006, pp. 207–221.

[15] T. Bloom, "Evaluating Synchronization Mechanisms," in *Proc. of Symposium on Operating Systems Principles*, 1979, pp. 24–32.

[16] B. Bogaert and S. Tison, "Equality and Disequality Constraints on Direct Subterms in Tree Automata," in *Proc. of the Symposium on Theoretical Aspects of Computer Science (STACS)*, 1992, pp. 161–171.

[17] A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar, "Programs with Lists Are Counter Automata," in *Proc. of Computer Aided Verification (CAV)*, 2006, pp. 517–531.

[18] A. Bouajjani, J. Esparza, and O. Maler, "Reachability Analysis of Pushdown Automata: Application to Model-Checking," in *Proc. of the International Conference on Concurrency Theory (CONCUR)*, 1997, pp. 135–150.

[19] A. Bouajjani, P. Habermehl, P. Moro, and T. Vojnar, "Verifying Programs with Dynamic 1-Selector-Linked Structures in Regular Model Checking," in *Proc. of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2005, pp. 13–29.

[20] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar, "Abstract Regular Tree Model Checking of Complex Dynamic Data Structures," in *Proc. of Static Analysis Symposium (SAS)*, 2006, pp. 52–70.

[21] C. Bouillaguet, V. Kuncak, T. Wies, K. Zee, and M. Rinard, "Using First-order Theorem Provers in the Jahob Data Structure Verification System," in *Proc. of Verfication Model Checking and Abstract Interpretation (VMCAI)*, 2007, pp. 74–88.

[22] C. Boyapati, R. Lee, and M. Rinard, "Ownership Types for Safe Programming: Preventing Data Races and Deadlocks," in *Proc. of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2002, pp. 211–230.

[23] R. E. Bryant, "Symbolic Simulation – Techniques and Applications," in *Proc. of ACM/IEEE Design Automation Conference (DAC)*, 1990, pp. 517–521.

[24] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang, "Beyond Reachability: Shape Abstraction in the Presence of Pointer Arithmetic," in *Proc. of Static Analysis Symposium (SAS)*, 2006, pp. 182–203.

[25] O. Carton, "Chain Automata," *Theoretical Computer Science*, vol. 161, no. 1-2, pp. 191–203, Jul. 1996.

[26] D. Caucal, "On the Regular Structure of Prefix Rewriting," in *Colloquium on Trees in Algebra and Programming*, 1990, pp. 87–102.

[27] E. Clarke, E. A. Emerson, S. Jha, and A. Sistla, "Symmetry reductions in model checking," in *Proc. of Computer Aided Verification (CAV)*, 1998, pp. 147–158.

[28] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded Model Checking Using Satisfiability Solving," *Formal Methods in System Design (FMSD)*, vol. 19, no. 1, pp. 7–34, Jul. 2001.

[29] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-Guided Abstraction Refinement," in *Proc. of Computer Aided Verification (CAV)*, 2000, pp. 154–169.

[30] E. M. Clarke and E. A. Emerson, "Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic," in *Logics of Programs*, 1981, pp. 52–71.

[31] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi, "Tree Automata Techniques and Applications," Available on: http://www.grappa.univ-lille3.fr/tata, 2007, release October, 12th 2007.

[32] J. C. Corbett, "Evaluating Deadlock Detection Methods for Concurrent Software," *IEEE Trans. on Software Engineering*, vol. 22, no. 3, p. 161180, 1996.

[33] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby, "Bandera: a Source-level Interface for Model Checking `Java` Programs," in *Proc. of Interna-*

tional Conference on Software Engineering (ICSE), 2000, pp. 762–765.

[34] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. MIT Press, 2001.

[35] D. Dams and K. Namjoshi, "Shape Analysis through Predicate Abstraction and Model Checking," in *Proc. of Verfication Model Checking and Abstract Interpretation (VMCAI)*, 2003, pp. 310–323.

[36] B. A. Davey and H. A. Priestley, *Introduction to Lattices and Order*, 2nd ed. Cambridge University Press, May 2002.

[37] L. de Moura and N. B. rner, "Z3: An Efficient SMT Solver," in *Proc. of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008, pp. 337–340.

[38] W. de Roever, F. de Boer, U. Hanneman, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers, *Concurrency Verification: Introduction to Compositional and Non-Compositional Methods*, 1st ed. Cambridge University Press, Jan. 2001.

[39] C. DeMartini, R. Iosif, and R. Sisto, "A Deadlock Detection Tool for Concurrent `Java` Programs," *Software Practice & Experience*, vol. 29, no. 7, pp. 577–603, 1999.

[40] J. V. Deshmukh, E. A. Emerson, and S. Sankaranarayanan, "Symbolic Deadlock Analysis in Concurrent Libraries and Their Clients," in *Proc. of Automated Software Engineering (ASE)*, 2009, pp. 480–491.

[41] J. V. Deshmukh and E. A. Emerson, "Verification of Recursive Methods on Tree-like Data Structures," in *Proc. of Formal Methods in Computer-Aided Design (FMCAD)*, 2009, pp. 33–40.

[42] J. V. Deshmukh, E. A. Emerson, and P. Gupta, "Automatic Verification of Parameterized Data Structures," in *Proc. of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2006, pp. 27–41.

[43] J. V. Deshmukh, E. A. Emerson, and R. Samanta, "Economical Transformations of Structured Data," Dept. of Computer Science, University of Texas at Austin, Tech. Rep., 2010, publication # TR-10-28.

[44] J. V. Deshmukh, G. Ramalingam, V. Ranganath, and K. Vaswani, "Logical Concurrency Control from Sequential Proofs," in *Proc. of European Symposium on Programming (ESOP)*, 2010, pp. 226–245.

[45] D. Distefano, P. W. O'Hearn, and H. Yang, "A Local Shape Analysis Based on Separation Logic," in *Proc. of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2006, pp. 287–302.

[46] V. D'Silva, D. Kroening, and G. Weissenbacher, "A Survey of Automated Techniques for Formal Software Verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1165–1178, Jul. 2008.

[47] B. Dutertre and L. de Moura, "A Fast Linear-Arithmetic Solver for DPLL(T)," in *Proc. of Computer Aided Verification (CAV)*, 2006, pp.

81–94.

[48] ——, "The YICES SMT solver," Aug. 2006, published: Tool paper at http://yices.csl.sri.com/tool-paper.pdf.

[49] E. A. Emerson, "Automata, Tableaux, and Temporal Logics," in *Logics of Programs*, 1985, pp. 79–88.

[50] ——, "Uniform Inevitability is Tree Automation Ineffable," *Information Processing Letters*, vol. 24, no. 2, pp. 77–79, 1987.

[51] ——, "Automated Temporal Reasoning about Reactive Systems," in *Proc. of the Banff Higher Order Workshop on Logics for Concurrency: Structure versus Automata*, 1996, pp. 41–101.

[52] E. A. Emerson and E. M. Clarke, "Characterizing Correctness Properties of Parallel Programs Using Fixpoints," in *Proc. of the International Colloquium on Automata, Languages and Programming (ICALP)*, 1980, pp. 169–181.

[53] E. A. Emerson, S. Jha, and D. Peled, "Combining partial order and symmetry reductions," in *Proc. of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 1997, pp. 19–34.

[54] E. A. Emerson and C. S. Jutla, "Tree automata, Mu-Calculus and determinacy," in *Proc. of Foundations of Computer Science (FoCS)*, 1991, pp. 368–377.

[55] ——, "The Complexity of Tree Automata and Logics of Programs," *SIAM J. Comput.*, vol. Vol. 29, pp. 132–158, 2000.

[56] E. A. Emerson and C. Jutla, "The complexity of Tree Automata and Logics of Programs," in *Proc. of Foundations of Computer Science (FoCS)*, vol. 0, 1988, pp. 328–337.

[57] E. A. Emerson and V. Kahlon, "Model Checking Large-Scale and Parameterized Resource Allocation Systems," in *Proc. of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2002, pp. 55–69.

[58] ——, "Model Checking Guarded Protocols," in *Proc. of Logic In Computer Science (LICS)*, 2003, pp. 361–370.

[59] E. A. Emerson and A. P. Sistla, "Symmetry and Model Checking," *Formal Methods in System Design (FMSD)*, vol. 9, no. 1, pp. 105–131, 1996.

[60] E. A. Emerson and T. Wahl, "Dynamic Symmetry Reduction," in *Proc. of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2005, pp. 382–396.

[61] J. Engelfriet and H. Hoogeboom, "Tree-walking Pebble Automata," *Jewels Are Forever, Contributions to Theoretical Computer Science in honor of Arto Salomaa*, vol. 1644, pp. 72–83, 1999.

[62] D. Engler and K. Ashcraft, "RacerX: effective, static detection of race conditions and deadlocks," *ACM SIGOPS Operating System Review*, vol. 37, no. 5, pp. 237–252, Dec. 2003.

[63] J. Esparza and J. Knoop, "An Automata-Theoretic Approach to Interprocedural Data-Flow Analysis," in *Proc. of Foundations of Software Science and Computation Structures (FOSSACS)*, 1999, pp. 14–30.

[64] A. Finkel, E. D. Cachan, B. Willems, and P. Wolper, "A Direct Symbolic Approach to Model Checking Pushdown Systems (Extended Abstract)," *Elec. Notes Theor. Comput. Sci.*, vol. 9, pp. 27–37, 1997.

[65] A. Finkel, E. Lozes, and A. Sangnier, "Towards Model-Checking Programs with Lists," in *Infinity in Logic and Computation*, 2009, pp. 56–86.

[66] C. Flanagan, S. N. Freund, and J. Yi, "Velodrome: a Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs," in *Proc. of Programming Language Design and Implementation (PLDI)*, 2008, pp. 293–303.

[67] C. Flanagan and P. Godefroid, "Dynamic Partial-order Reduction for Model Checking Software," *ACM SIGPLAN Notices*, vol. 40, no. 1, pp. 110–121, 2005.

[68] V. K. Garg, *Concurrent and Distributed Computing in Java*. Wiley Interscience, 2004.

[69] S. Ginsburg, S. Greibach, and M. Harrison, "One-way Stack Automata," *Journal of the ACM (JACM)*, vol. 14, no. 2, pp. 389–418, 1967.

[70] ——, "Stack Automata and Compiling," *Journal of the ACM (JACM)*, vol. 14, no. 1, pp. 172–201, 1967.

[71] N. Globerman and D. Harel, "Complexity Results for Two-way and Multi-Pebble Automata and their Logics," *Theoretical Computer Science*, vol. 169, no. 2, pp. 161–184, 1996.

[72] A. Gotsman, J. Berdine, and B. Cook, "Interprocedural Shape Analysis with Separated Heap Abstractions," in *Proc. of Static Analysis Symposium (SAS)*, 2006, pp. 240–260.

[73] B. Gulavani, S. Chakraborty, G. Ramalingam, and A. Nori, "Bottom-Up Shape Analysis," in *Proc. of Static Analysis Symposium (SAS)*, 2009, pp. 188–204.

[74] A. Gupta, "Formal Hardware Verification Methods: A Survey," *Formal Methods in System Design*, vol. 1, no. 2-3, pp. 151–238, 1992.

[75] P. Habermehl, R. Iosif, A. Rogalewicz, and T. Vojnar, "Proving Termination of Tree Manipulating Programs," in *Proc. of Automated Technology for Verification and Analysis (ATVA)*, 2007, pp. 145–161.

[76] P. Habermehl, R. Iosif, and T. Vojnar, "Automata-Based Verification of Programs with Tree Updates," in *Proc. of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2006, pp. 350–364.

[77] B. K. Haddon, "Nested Monitor Calls," *SIGOPS Oper. Syst. Rev.*, vol. 11, no. 4, pp. 18–23, 1977.

[78] D. Harel and D. Raz, "Deciding Emptiness for Stack Automata on Infinite Trees," *Information and Computation*, vol. 113, no. 2, pp. 278–299, 1994.

[79] K. Havelund, "Using Runtime Analysis to Guide Model Checking of Java Programs," in *Proc. of* SPIN *Workshop on Model Checking of Software (SPIN)*, 2000, pp. 245–264.

[80] K. Havelund and J. S. k, "Applying Model Checking in Java Verification," in *Theoretical and Practical Aspects of* SPIN *Model Checking, 5th and 6th International* SPIN *Workshops (SPIN)*, 1999, pp. 216–231.

[81] K. Havelund and T. Pressburger, "Model Checking `Java` Programs using `Java PathFinder`," *Software Tools for Technology Transfer (STTT)*, vol. 2, no. 4, p. 366381, Mar. 2000.

[82] T. A. Henzinger, R. Jhala, and R. Majumdar, "Permissive Interfaces," in *Proc. of the European Software Engineering Conference held jointly with the Foundations of Software Engineering (ESEC/FSE)*, 2005, pp. 31–40.

[83] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[84] G. J. Holzmann, *The* SPIN *Model Checker*. Addison-Wesley, 2003.

[85] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Pearson/Addison Wesley, 2007.

[86] J. E. Hopcroft and J. D. Ullman, *Formal Languages and their Relation to Automata*. Addison-Wesley Pub. Co., Jan. 1969.

[87] S. Hudson, F. Flannery, and C. S. Ananian, "LALR Parser Generator in `Java`," http://www2.cs.tum.edu/projects/cup/.

[88] D. Jackson, "Alloy: a Lightweight Object Modelling Notation," *ACM Trans. on Soft. Eng. and Meth. (TOSEM)*, vol. 11, no. 2, p. 290, 2002.

[89] Jajuk Advanced Jukebox, "Bug Ticket #850," http://trac.jajuk.info/ticket/850, 2008.

[90] B. Jeannet, A. Loginov, T. Reps, and M. Sagiv, "A Relational Approach to Interprocedural Shape Analysis," *ACM Trans. Program. Lang. Syst. (TOPLAS)*, vol. 32, no. 2, pp. 1–52, 2010.

[91] N. D. Jones and S. S. Muchnick, *Program Flow Analysis: Theory and Applications*. Prentice Hall, Jun. 1981.

[92] V. Kahlon, "Boundedness vs. Unboundedness of Lock Chains: Characterizing Decidability of Pairwise CFL-Reachability for Threads Communicating via Locks," in *Proc. of Logic In Computer Science (LICS)*, 2009, pp. 27–36.

[93] V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta, "Fast and Accurate Static Data-race Detection for Concurrent Programs," in *Proc. of Computer Aided Verification (CAV)*, 2007, pp. 226–239.

[94] S. Khurshid, M. Malik, and E. Uzuncaova, "An Automated Approach for Writing Alloy Specifications Using Instances," in *Proc. of International Symposium on Leveraging Applications of Formal Methods (ISoLA)*, 2006, pp. 449–457.

[95] G. Klein, "The Fast Scanner Generator for `Java`," http://jflex.de/index.html.

[96] O. Kupferman, N. Piterman, and M. Vardi, "Pushdown specifications," in *Proc. of Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, 2002, pp. 262–277.

[97] O. Kupferman and A. Pnueli, "Once and For All," in *Proc. of Logic in Computer Science (LICS)*, 1995, p. 25.

[98] S. K. Lahiri and M. Musuvathi, "An Efficient Decision Procedure for UTVPI Constraints," in *Proc.* 5^{th} *International Workship on Frontiers of Combining Systems*, 2005, p. 168183.

[99] P. Lam, V. Kuncak, and M. Rinard, "Hob: A Tool for Verifying Data Structure Consistency," in *Proc. of Compiler Construction (CC)*, 2005, pp. 237–241.

[100] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[101] T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm, "Putting Static Analysis to Work for Verification: A Case Study," *ACM SIGSOFT Softw. Eng. Notes*, vol. 25, no. 5, pp. 26–38, 2000.

[102] L. Li and C. Verbrugge, "A Practical MHP Information Analysis for Concurrent `Java` Programs," in *Proc. of the Workshop on Languages and Compilers for Parallel Computing*, 2004, pp. 194–208.

[103] A. Lister, "The Problem of Nested Monitor Calls," *SIGOPS Oper. Syst. Rev.*, vol. 11, no. 3, pp. 5–7, 1977.

[104] M. Malik, A. Pervaiz, and S. Khurshid, "Generating Representation Invariants of Structurally Complex Data," in *Proc. of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2007, pp. 34–49.

[105] M. Z. Malik, A. Pervaiz, E. Uzuncaova, and S. Khurshid, "Deryaft: a Tool for Generating Representation Invariants of Structurally Complex Data," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2008, pp. 859–862.

[106] R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv, "Predicate Abstraction and Canonical Abstraction for Singly-Linked Lists," in *Proc. of*

Verfication Model Checking and Abstract Interpretation (VMCAI), 2005, pp. 181–198.

[107] K. L. McMillan, *Symbolic Model Checking*, 1st ed. Kluwer Academic, Jul. 1993.

[108] A. Møller and M. I. Schwartzbach, "The Pointer Assertion Logic Engine," in *Proc. of PLDI*, 2000, pp. 221—231.

[109] M. Naik, A. Aiken, and J. Whaley, "Effective Static Race Detection for `Java`," in *Proc. of Programming Language Design and Implementation (PLDI)*, 2006, pp. 308–319.

[110] M. Naik, C. Park, K. Sen, and D. Gay, "Effective Static Deadlock Detection," in *Proc. of the International Conference on Software Engineering (ICSE)*, 2009, pp. 386–396.

[111] F. Neven, "Automata, logic, and XML," in *Proc. of Computer Science Logic*, 2002, pp. 671–711.

[112] F. Neven, T. Schwentick, and V. Vianu, "Towards Regular Languages over Infinite Alphabets," *Mathematical Foundations of Computer Science*, pp. 560–572, 2001.

[113] ——, "Finite State Machines for Strings over Infinite Alphabets," *ACM Trans. Comput. Logic*, vol. 5, no. 3, pp. 403–435, 2004.

[114] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Springer, Oct. 1999.

[115] P. W. O'Hearn, J. C. Reynolds, and H. Yang, "Local Reasoning about Programs that Alter Data Structures," in *Proc. of the 15th International Workshop on Computer Science Logic*, 2001, pp. 1–19.

[116] Open Source Mail Archive, "Message #150," http://osdir.com/ml/java.hsqldb.user/2004-03/msg00150.html, 2004.

[117] ——, "Bug 159," http://osdir.com/ml/java.openjdk.distro-packaging.devel/2008-06/msg00061.html, 2008.

[118] F. Otto and T. Moschny, "Finding Synchronization Defects in Java Programs: Extended Static Analyses and Code Patterns," in *Proc. of* 1^{st} *International Workshop on Multicore Software Engineering*, 2008, pp. 41–46.

[119] D. Peled, "Combining Partial Order Reductions with On-the-fly Model-Checking," *Formal Methods in System Design (FMSD)*, vol. 8, no. 1, pp. 39–64, Jan. 1996.

[120] W. Peng and S. Iyer, "A New Type of Pushdown Automata on Infinite Trees," *Foundations of Computer Science (FoCS)*, vol. 6, pp. 169–186, 1995.

[121] S. Pinter and P. Wolper, "A Temporal Logic for Reasoning about Partially Ordered Computations," in *Proceedings of Principles of Distributed Computing (PODC)*, 1984, p. 37.

[122] M. Prasad, A. Biere, and A. Gupta, "A Survey of Recent Advances in SAT-Based Formal Verification," *Software Tools for Technology Transfer*, vol. 7, no. 2, pp. 156–173, 2005.

[123] J. Queille and J. Sifakis, "Specification and verification of concurrent systems in CESAR," in *Proceedings of the 5th Colloquium on International Symposium on Programming*. Springer-Verlag, 1982, pp. 337–351.

[124] G. Ramalingam, "Context-Sensitive Synchronization-Sensitive Analysis is Undecidable," *ACM Trans. Program. Lang. Syst. (TOPLAS)*, vol. 22, no. 2, pp. 416–430, 2000.

[125] J. C. Reynolds, "Separation Logic: A Logic for Shared Mutable Data Structures," in *Proc. of Logic in Computer Science (LICS)*, 2002, pp. 55–74.

[126] N. Rinetzky and M. Sagiv, "Interprocedural Shape Analysis for Recursive Programs," in *Proc. of Compiler Construction (CC)*, 2001, pp. 133–149.

[127] N. Rinetzky, M. Sagiv, and E. Yahav, "Interprocedural Shape Analysis for Cutpoint-Free Programs," in *Proc. of Static Analysis Symposium (SAS)*, 2005, pp. 284–302.

[128] A. Rogalewicz, "Verification of Programs with Complex Data Structures," Ph.D. dissertation, Brno University of Technology, Czech Republic, 2007.

[129] A. L. Rosenberg, "On Multi-Head Finite Automata," *IBM J. Res. Dev.*, vol. 10, no. 5, pp. 388–394, 1966.

[130] M. Sagiv, T. Reps, and R. Wilhelm, "Solving Shape-Analysis Problems in Languages with Destructive Updating," *Transactions on Programming Languages and Systems*, vol. 20, no. 1, pp. 1–50, jan 1998.

[131] ——, "Parametric Shape Analysis via 3-valued Logic," in *Proc. of Principles of Programming Languages (POPL)*, 1999, pp. 105–118.

[132] R. Samanta, J. V. Deshmukh, and E. A. Emerson, "Automatic Generation of Local Repairs for Boolean Programs," in *Proc. of Formal Methods in Computer-Aided Design (FMCAD)*, 2008, pp. 213–222.

[133] P. Schnoebelen, "The Complexity of Temporal Logic Model Checking," *Advances in Modal Logic*, vol. 4, pp. 437–459, 2003.

[134] S. Schwoon, "Model-checking pushdown systems," Ph.D. dissertation, Technische Universität München, Universitätsbibliothek, 2002.

[135] V. K. Shanbhag, "Deadlock-Detection in Java-Library using Static-Analysis," in *Proc. of the 15th Asia-Pacific Software Engineering Conference*, 2008, pp. 361–368.

[136] M. Sharir and A. Pnueli, "Two Approaches to Interprocedural Data Flow Analysis," in *Program Flow Analysis: Theory and Applications*, S. S. Muchnick and N. D. Jones, Eds. Prentice Hall International, 1981, pp. 189–234.

[137] M. Sipser, *Introduction to the Theory of Computation*, 1st ed. Thomson Course Technology, 2006, Dec. 1996.

[138] K. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson, "Software Assurance by Bounded Exhaustive Testing," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2004, pp. 133–142.

[139] Sun Developer Network Bug Database, http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=xxxx, 2007, bug-id provided at citation.

[140] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co, "Soot - a Java Optimization Framework," in *Proc. of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, 1999, p. 125135.

[141] J. P. van Best, "Tree-Walking Automata and Monadic Second Order Logic," Master's thesis, Leiden University, July 1998.

[142] M. Češka, P. Erlebach, and T. Vojnar, "Pattern-Based Verification of Programs with Extended Linear Linked Data Structures," *Elec. Notes Theor. Comput. Sci.*, vol. 145, pp. 113–130, Jan. 2006.

[143] ——, "Pattern-Based Verification for Trees," in *Computer Aided Systems Theory EUROCAST*, 2007, pp. 488–496.

[144] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model Checking Programs," *Automated Software Engineering*, vol. 10, no. 2, pp. 203–232, Apr. 2003.

[145] C. von Praun, "Detecting Synchronization Defects in Multi-Threaded Object-Oriented Programs," Ph.D. dissertation, ETH Zurich, 2004.

[146] J. Whaley, M. C. Martin, and M. S. Lam, "Automatic Extraction of Object-oriented Component Interfaces," *ACM SIGSOFT Soft. Eng. Notes*, vol. 27, pp. 218–228, 2002.

[147] T. Wies, V. Kuncak, K. Zee, M. Rinard, and A. Podelski, "On Verifying Complex Properties using Symbolic Shape Analysis," in *Workshop on Heap Abstraction and Verification*, 2006.

[148] A. Williams, W. Thies, and M. D. Ernst, "Static Deadlock Detection for Java Libraries," in *Proc. of the European Conference on Object-Oriented Programming (ECOOP)*, 2005, pp. 602–629.

[149] P. Wolper, "Expressing Interesting Properties of Programs in Propositional Temporal Logic," in *Proc. of Principles of Programming Languages (POPL)*, 1986, pp. 184–193.

[150] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn, "Scalable Shape Analysis for Systems Code," in *Proc. of Computer Aided Verification (CAV)*, 2008, pp. 385–398.

[151] J. Yang and C. Seger, "Generalized Symbolic Trajectory EvaluationAbstraction in Action," in *Proc. of Formal Methods in Computer-Aided Design (FMCAD)*, 2002, pp. 70–87.

[152] G. Yorsh, A. Rabinovich, M. Sagiv, A. Meyer, and A. Bouajjani, "A Logic of Reachable Patterns in Linked Data-Structures," in *Proc. of Foundations of Software Science and Computation Structures (FOSSACS)*, 2006, pp. 94–110.

[153] F. Zaraket, A. Aziz, and S. Khurshid, "Sequential Circuits for Relational Analysis," in *International Conference on Software Engineering (ICSE)*, 2007, p. 1322.

[154] K. Zee, V. Kuncak, and M. Rinard, "Verifying Linked Data Structure Implementations," in *Proc. of International Symposium on Parallel and Distributed Processing*, 2008, pp. 1–5.

[155] K. Zee, V. Kuncak, and M. C. Rinard, "Full Functional Verification of Linked Data Structures," in *Proc. of Programming Language Design and Implementation (PLDI)*, 2008.

Vita

Jyotirmoy Vinay Deshmukh was born in 1979 in Mumbai, India to Nandini Vinay Deshmukh and Vinay Dattatraya Deshmukh. He completed his schooling from the I.E.S. Modern English School in 1994, and the D. G. Ruparel Junior College in 1996, both in Mumbai. From 1996 to 2000, he studied at the Veermata Jijabai Technological Institute (V.J.T.I.) in Mumbai, obtaining a Bachelor's degree in Engineering (B.E.) with a specialization in Electronics. After a brief tenure as a Circuit and Software Design Engineer at Texas Instruments India, he joined the graduate program at the University of Texas at Austin in 2002. He was named a 2010 Computing Innovation Fellow by the Computing Research Association (CRA) and the Computing Community Consortium (CCC). With this fellowship, he will continue his post-doctoral research work at the University of Pennsylvania.

Permanent address: 8, Ramkripa,
D.L. Vaidya Road,
Dadar, Mumbai, India 400028

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.