LOGICS AND ALGORITHMS FOR SOFTWARE MODEL CHECKING

Swarat Chaudhuri

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

2007

Rajeev Alur
Supervisor of Dissertation

Rajeev Alur
Graduate Group Chairperson

For my parents, Drs. Asru and Jayasri Chaudhuri

Acknowledgements

A Ph.D. is as much of a journey as it is a degree. Luckily for me, as I went farther and farther in this journey, I felt more and more excited by my research, and at the point where I am now, my work is far more to me than just work. Not only do I look at my years at Penn with affection and nostalgia, but I count them among the best years of my life. This would not be so without the guidance and support of many others.

First of all, I would like to thank my advisor Rajeev Alur, who has taught me in more ways than he will acknowledge himself. I continue to be amazed by the way his technical acumen combines with his eye for clean formulations and the central point, and his imprint on the way I identify, formulate, and solve problems is enormous. He has also been a kind and effective advisor, allowing me a great amount of freedom while being actively involved in my research and nudging me in the right directions. I will be applying lessons I have learnt from him all through the academic career I am about to start.

Many thanks go to Madhusudan Parthasarathy, my co-author in a large part of the work that went into this thesis. I am also grateful to Stephen Fink, Satish Chandra, and Eran Yahav of IBM T. J. Watson Research Center, Hawthorne, my summer with whom was enormously educative and fun. Thanks also to David Melski and Kousha Etessami for guiding me during my summers at Grammatech Corporation and Lucent Bell Laboratories. I am also grateful to Val Tannen, Jean Gallier, Andre Scedrov, and Mihalis Yannakakis for being kind committee members, and the faculty and staff at Penn's computer science department for providing a cosy academic atmosphere. A special cheer goes to Sudipto Guha, for all the jokes, avuncular advice, and algorithmic insights.

A section like this would be incomplete unless I mention my office-mates and friends Pavol Cerny and Mikhail Bernadsky. I thank them for our many academic discussions, our numerous lunches and dinners together, and all the time we have spent discussing philosophy, politics and the future of computer science.

This is also the time to express my gratitude to all my friends who have been there for me during my graduate student years. I am grateful to you for sharing my ups and downs, for the encouragement and affection, movies and squash games—Philadelphia would not be what is to me without you. In particular, I would like to thank Cheryl Testa, Sumantra Sen, Justin Walters, Kandaswamy Vijayan, and Alwyn Goodloe. A lot of gratitude is reserved for Tina Peterson, who, aside from being a good media theorist, makes my life get better all the time.

Last but never the least, I would like to thank my family in India for helping me get where I am today. Most of all, I am grateful to my parents, who have loved and inspired me all through my life. From an early age, they drilled in me a love for inquiry, reason, science, and research, and encouraged me to dream big, love exploration, and keep an open mind. Many years later, they still urge me to dream and explore as I follow their footsteps into academe. It is to them that I dedicate this thesis.

ABSTRACT

LOGICS AND ALGORITHMS FOR SOFTWARE MODEL CHECKING

Swarat Chaudhuri

Rajeev Alur

Software model checking, an algorithmic, specification-driven approach to software analysis, has emerged as an active area of research in the last few years, producing a number of successful tools. The central question here is: does a model of a procedural program (typically a context-sensitive or pushdown abstraction) satisfy its requirements (typically expressed using temporal logics or automata)? In this thesis, we study specification formalisms and analysis algorithms applicable to this problem.

We start by observing that classical temporal logics like the μ-calculus or CTL cannot specify "context-sensitive" requirements such as: "If a lock is not held before a call, it must be released before the matching return." A fix, we show, is to model branching behaviors of programs not by computation trees, but by infinite graphs called nested trees. Logics and automata interpreted on these new structures are now defined, and the model checking problem is phrased as: "Does the nested tree generated by a program satisfy a property?" This formulation lets us specify context-sensitive requirements such as pre/post-conditions, "local" dataflow properties, and stack-sensitive access control requirements, and, generally, leads to more modular specifications. Yet, these formalisms are theoretically robust, the complexity of model checking stays the same as before, and symbolic model checking is possible. An application is a specification language (called Pal) that can modularly state context-sensitive safety specifications, and has use in model checking and runtime monitoring of C programs.

On the algorithmic end, we study reachability in recursive state machines (or equivalently, pushdown systems), a problem central to software model checking and program analysis that was, for long, believed to intrinsically cubic and behind the "cubic bottleneck" of its many applications. We show that we can solve this problem in subcubic time by a simple adaptation of a known technique. We also show that a better algorithm exists if the input machine does not have infinite recursive loops— this algorithm is obtained via a new algorithm for graph transitive closure that is possibly of independent interest— and identify a gradation in the complexity of the pushdown reachability problem as recursion is restricted.

Note: *The work in this thesis also appears in the following four articles:*

- Rajeev Alur, Swarat Chaudhuri, and P. Madhusudan. A fixpoint calculus for local and global program flows. In *33rd Annual Symposium on Principles of Programming Languages (POPL)*, 2006.

- Rajeev Alur, Swarat Chaudhuri, and P. Madhusudan. Languages of nested trees. In *18th International Conference on Computer-Aided Verification (CAV)*, 2006.

- Swarat Chaudhuri and Rajeev Alur. Instrumenting C programs with nested word monitors (Tool paper). In *14th Workshop on Model Checking Software (SPIN)*, 2007.

- Swarat Chaudhuri. Subcubic algorithms for recursive state machines. In *35th Annual Symposium on Principles of Programming Languages (POPL)*, 2008.

Contents

List of Figures

Chapter 1

Introduction

1.1 Reasoning about correctness

Since the 1960s, formally proving programs correct has been a grand challenge for computer science. While the proof systems that Floyd and Hoare [Flo67, Hoa69] designed in their seminal papers could in principle do such proofs, deploying them in practice was difficult. As manual proofs were untenable for programs beyond a certain size, efforts to partially automate deductive proofs were launched. In this vision, a programmer proves a program correct using a *proof assistant* that checks proofs and supplies hints for proof construction. Over the last few decades, many heuristics and decision procedures for deduction have emerged in this area, known as *deductive verification* [ORR$^+$96, KMM00], leading to proofs of correctness of several large and interesting software systems. However, proving programs correct in this manner remains a herculean task, requiring enormous levels of expertise and effort.

A different set of methods, known collectively as *program analysis* [NNH99], arose from the programming language community and stressed approximate solutions. Sample questions here were: "What is a conservative approximation of the set of variables a pointer variable may point to?" or "Is the variable `x` live at a certain program location?" or "Is a certain memory location freed along paths through the

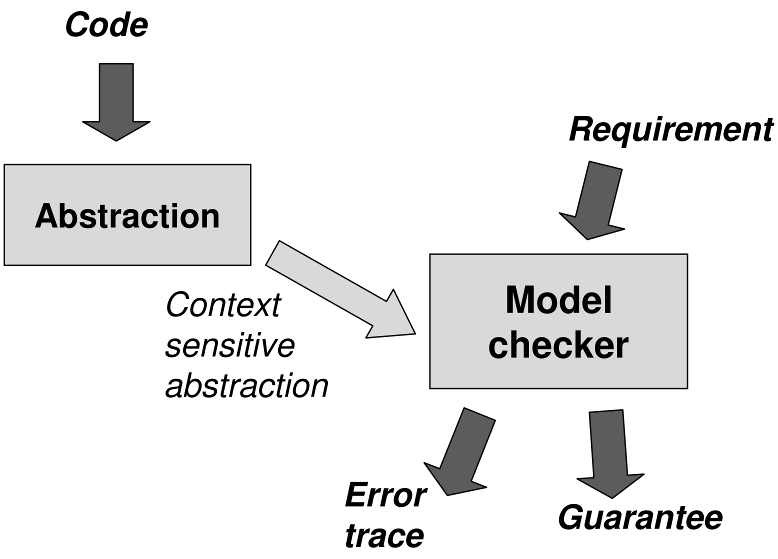

Figure 1.1: Software model checking

current procedure?" Some of these tools aimed to infer information about dataflow and memory use in programs for compile-time error detection; some were used to optimize code and were invisible to the programmer. Practice preceded theory in program analysis, and typically, scalability was considered more important than depth of analysis. However, the theory of *abstract interpretation* [CC77] offers solid foundations for sound (but incomplete) program analysis.

A third approach to reasoning about programs is *model checking* [CE81, QS82, CGP99]. While deductive verification is proof-theoretic, this approach is language-theoretic and involves answering a question of the type: "Does the temporal behavior of a program belong to a formal language of " specified" behaviors?" Concretely, a program's evolution over time is modeled by a logical structure. For example, we could use an infinite tree to model a program's branching behavior— each path in this tree models a possibly infinite execution. A specification is a formal language of "correct behaviors" given, typically, by an automaton or a formula in a logic. For example, if a tree is used to model a program's unfolding, a specification logic typically states properties such as: "on every path in the tree from a node satisfying a proposition *lock*, there is a node satisfying a proposition *unlock*" (this corresponds to the requirement: "every lock acquired by the program is eventually released"). The corresponding formal language is the set of trees satisfying this formula. The model checking problem is then: "do the logical structures generated by a given program P belong to a language L?" The key idea in model checking is that the specification language and its models are sufficiently restricted to make this question decidable, so that algorithmic (typically search-based) methods can be applied. This raises a number of questions: how far can the expressiveness of a logic be pushed without sacrificing decidability? How much of this expressiveness is needed to capture realistic specifications? Is the complexity manageable? How can we design better algorithms and data structures for model checking?

The initial application of model checking was to verify finite-state programs describing hardware [McM93] or network protocols [Hol91]. Specifications were usually in linear or branching-time *propositional temporal logic* [Pnu77], which can express properties such as: "along all paths in the program, the proposition p holds until a state satisfying q is reached," or "p is true at infinitely many points on some program path." The model-checking problems for these logics (e.g., LTL, CTL, the μ-calculus) were found to be decidable, though the use of novel data structures (e.g., the use of binary decision diagrams) and search techniques (e.g., on-the-fly traversal) were needed to make verification applicable to real-life circuits and protocols, where the number of program states can be enormous (the so-called *state explosion problem*).

Research on temporal logic model checking had a symbiotic relationship with theoretical research on logics and automata. For example, the decidability of temporal logics on word and tree models are really special cases of Büchi and Rabin's seminal results [Büc62, Rab69] on the decidability of monadic second order logic (MSO) on words and trees. There are many deep connections between theories of logic, automata and games [Tho97, GTW02]— for instance, MSO logic (on words, trees) is a logical characterization of finite-state automata (on words, trees), the μ-calculus has polynomial translations to and from bisimulation-closed alternating automata on trees [EJ91, JW96], parity games and their zero-memory determinacy provide crucial steps that simplify the decidability proof, etc. On one hand, these results have applications in practical algorithms in model checking—such as the automata-theoretic procedures implemented in the SPIN model checker. On the other hand, inspired by model checking, new studies on expressiveness and decidability of logics were launched.

1.2 Context-sensitive model checking

Because of concerted research over the last twenty-five years, hardware model checking has by now become a mature technology that is used regularly in the industry. Inspired by its success, researchers in formal methods have tried to apply model checking to general software. For nearly all useful classes of software, this is an undecidable problem, and even if we accept non-termination or one-sided errors, the problem remains highly challenging. Programs written in a modern programming language have many complex features such as the stack, the heap, pointer aliases, integer and floating point operations, and concurrent execution, reasoning about which in any automated manner is a challenge.

Some progress has, however, been made in the last few years. A popular and successful idea is to use *data abstraction* in conjunction with model checking. Instead of analyzing a program P directly, analysis is now applied to an abstraction or over-approximation $\widehat{P}$ that only has bounded data variables. Using techniques such as *predicate abstraction* [SH97, BMMR01], it is possible to obtain such an abstraction automatically. Now suppose we want to determine if every lock acquired in P is eventually released. Instead of analyzing P to obtain an answer to this question, we ask: "Does the abstraction $\widehat{P}$ release all the locks it acquires?" If $\widehat{P}$ satisfies this property, then so does P, and we can terminate knowing that we have proved a property of the original program. Otherwise, the model checker returns a sample path leading to the error state. Of course, because we are only analyzing an abstraction, this "counterexample" may be spurious. To overcome this problem, some software model checkers use an *abstraction-refinement loop* to automatically determine whether a counterexample is false, and if so, try model checking again with a more *refined* abstraction [CGJ$^+$00]. Of course, abstraction, counterexample checking, and refinement of software are all non-trivial problems, and will typically require the help of a theorem prover. Such techniques have been applied in the SLAM toolkit [BR01], which was developed to verify Microsoft device drivers and has since been transferred to driver writers as the Static Driver Verifier (SDV) [BCLR04]. This is one of the bigger success stories of software model checking.

A natural question is: what kind of abstractions do we pick? On one hand, the simpler our abstractions, the likelier checking is to be tractable. It is because of this that it is common in program analysis to stick to the simplest of program abstractions: the control-flow graphs. On the other hand, the simplest abstractions will usually not suffice if we insist on a reasonable degree of precision. For example, almost all modern software is procedural, and thus has control flow dependent on a *procedural context*—i.e., the state of a possibly unbounded call stack. Model checking an abstraction that does not model this *context-sensitive* control flow turns out to lead to many false counterexamples. As a result, it is common in this setting to abstract programs into *context-sensitive abstractions*. These systems can precisely capture the interprocedural control flow in a program written in a typical imperative language with recursive calls, while abstracting its data into a finite domain. While such a system has a finite set of control states, it has a potentially unbounded memory of past contexts on which future execution depends. Importantly, the history of contexts has a *nested* structure— if context A calls context B and C is called from within B, then control, on return from C, will go back to B first. Note that this nesting of contexts can be arbitrarily deep. An example abstraction of this sort is a recursive state machine [ABE$^+$05]— a collection of finite-state machines where each component machine can call other component machines recursively. Other models of provably equal expressiveness include pushdown systems [BEM97] and boolean programs [BR00].

Model checking such a system would involve verifying whether it satisfies certain linear or branching-time temporal requirements, specified using logics or automata. Questions of interest include: "Is a certain configuration reachable along an execution of the abstraction?" and "Is a certain control location involved in a cycle?" Answering these queries turns out to pose algorithmic challenges not arising in finite-state verification.

In this thesis we are interested in the question of *context-sensitive model checking for sequential, procedural programs.* More precisely, we ask the following questions:

1. *What are the "right" logics and automata usable in context-sensitive model checking?* The traditional answers here have been temporal logics and automata on words and trees. Do these notations, inherited from classical model checking, fail to express some important properties of context-sensitive program abstractions— for example, requirements involving the nesting of contexts? If so, then can we increase their expressiveness without losing out on tractability of model checking?

2. *Can we offer improvements over the existing algorithms for context-sensitive model checking?* Here we focus on the model checking problem for linear-time requirements on context-sensitive abstractions. For a fixed-size formula, this problem reduces to the control-state reachability problem for recursive state machines, or equivalently pushdown systems. Is the complexity of this problem

intrinsically cubic, as has long been believed, or is there an algorithm for it with a subcubic complexity?

In Part I of this dissertation, we address the former question, and in Part II, we tackle the latter. In the rest of this chapter, we give an overview of our methods and results.

1.3 Context-sensitive specifications

Requirements in model checking have traditionally been written using word or tree automata, or temporal logics such as LTL, CTL, CTL*, and the modal μ-calculus. Context-sensitive model checking for these formalisms are decidable— in fact, the problem is decidable even for monadic second-order (MSO) logic, a logic that is expressively equivalent to tree automata and includes all the above logics as fragments. However, all these formalisms suffer from a common shortcoming: they describe *regular* tree languages and cannot express "context-sensitive" requirements. Consider Hoare-style preconditions and postconditions [Hoa69]: "If p holds at a procedure call, then if the procedure terminates, q holds on return." Such temporal requirements on specific procedural contexts, rather than the global program execution, also arise in modern interface specification languages such as JML [BCC$^+$03]. Now, to express this requirement, we need to pair calls with matching returns. This, however, is beyond the expressive power of a finite-state tree automaton. Because of the connections between tree automata and temporal logics, neither can this requirement be expressed by a temporal logic like CTL or the μ-calculus.

A number of context-sensitive specifications also arise in software security. Take, for example, the access control requirement: "A method A should be invoked only if a privileged method B belongs to the call stack," checked at runtime in the Java or .NET stack inspection framework [WF98, NST01]. Or consider the inverse requirement: "If A has ever been on the stack, do not execute sensitive operation

X," or requirements that bound the size of the call stack to prevent stack overflow: "after any point where p holds, the number of interrupt-handlers in the call-stack should never exceed 5." As the stack of a program is potentially unbounded, these requirements cannot be expressed by classical temporal logics and automata.

Another source of context-sensitive properties is data flow analysis. In the program analysis literature, it has been argued that data flow analysis, such as the computation of live variables and very busy expressions, can be viewed as evaluation of μ-calculus or CTL formulas over abstractions of programs [Ste91, Sch98]. This correspondence does not hold when we need to account for *local* data flow paths. For instance, for an expression e that involves a variable local to a procedure P as well as a global variable, the set of control points within P at which e is very busy (that is, e is guaranteed to be used before any of its variables get modified) cannot be specified using a μ-calculus formula.

Further, in certain cases specifications that can reason about *branching* in addition to linear sequencing are needed. Consider the following property asserted at an entry u to a procedure of a program with *lock* and *unlock* methods: "Either (1) for all matching exits v of u, the resource X is locked at the time when control reaches v, or (2) X is unlocked at all these points." Here, the requirement on matching introduces context-sensitivity, and since locking or unlocking can happen along arbitrary points on paths from u from v, reasoning about multiple branches is warranted. Other examples include certain interprocedural data flow requirements, as well as pushdown reachability games that describe vulnerable "slices" in a program where some procedures are compromised by an attacker.

Now, individual static analysis techniques are available for many of these requirements [JMT99, CW02, EKS03, CMM$^+$04, BBFG04]. There are analyses for stack inspection-type properties, and interprocedural data flow analysis [RHS95] can compute data flow information involving local variables. The less understood phenomenon is the class of languages to which these properties correspond and the way they relate to each other. Is there a unified logical formalism that can connect all these seemingly disparate dots, extending the model-checking paradigm to properties such as above? Can we offer the programmer a flexible, decidable temporal logic or automaton model to write these requirements?

These are not merely academic questions. A key attraction of model checking from the practical point of view is that a programmer, once offered the freedom of a temporal specification language, can tailor her requirements about a program's behavior without getting lost in implementation details. A logic as above would extend this paradigm to interprocedural reasoning. Adding syntactic sugar to it, one could obtain domain-specific applications—for example, one can conceive of a language for security policies (even dynamic security policies) built on top of such a formalism.

At the same time, these questions open up avenues for interesting theoretical investigation. What would these logics look like in the first place? What kind of expressiveness will they have? Will they be decidable? Where would the trade-off between expressiveness and decidability lie?

1.3.1 Languages of nested trees

Traditionally, model-checking of branching-time requirements has involved determining whether the tree unfolding of the input program belongs to a tree language defined by the requirement. The first natural extension of this framework to requirements such as above is to consider *context-free tree languages* [Rou70, CDG$^+$02] as specification. Unfortunately, this causes the model-checking problem to become undecidable. The intuition behind the proof is that the emptiness problem for the intersection of two context-free languages—one encoding the program's control flow and the other the specification— is undecidable [HU79].

In this thesis, elaborating on work we have previously published as conference articles [ACM06a, ACM06b, CA07] and building up on similar efforts for the simpler linear-time setting [AM04, AM06], we identify an alternative phrasing of the

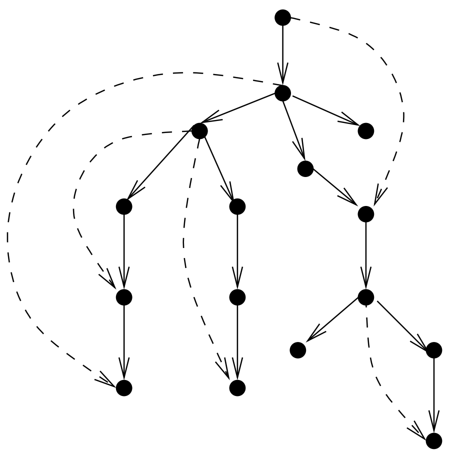

Figure 1.2: A nested tree

branching-time model checking problem that does not suffer from this problem. The key idea is to model a program's branching behavior not by a tree, but by a graph known as a *nested tree* that is similar to a tree but has some extra structure. More precisely, a nested tree is obtained by augmenting a tree (comprising a set of *tree edges*) with a set of properly nested *jump-edges* (see Figure 1.2; the jump-edges are dashed). Intuitively, these edges connect nodes modeling calls to their matching returns, and are the same as *summary edges* in interprocedural static analysis.

We show that context-sensitive program requirements can be phrased as *regular requirements on nested trees.* We establish this by developing a theory of *regular languages of nested trees*, which are accepted by finite-state automata and logics interpreted on nested trees. These formalisms can argue about succession along tree edges as well as jump-edges, generalizing automata and logics on trees. Since a matching node is a mere jump-successor in a nested tree, they are naturally capable of capturing context-sensitive program requirements such as pre/post conditions or for interprocedural data flow specifications. We also find that they have a robust theory and attractive computational properties, and can be directly applied to model checking.

1.3.2 A fixpoint calculus for nested trees

We start by introducing a fixpoint logic for infinite nested trees that is analogous to the modal μ-calculus for trees. The reader is reminded that the μ-calculus [Koz83, GTW02] is a modal logic with fixpoints interpreted over labeled transition systems, or equivalently, over their tree unfoldings. It is an extensively studied specification formalism with applications to program analysis, model checking, and database query languages [Eme90, Sti91]. From a theoretical perspective, its status as the *canonical* temporal logic for regular requirements is due to the fact that its expressiveness exceeds that of all commonly used temporal logics such as LTL, CTL, and CTL*, and equals that of *alternating parity tree automata* or the bisimulation-closed fragment of monadic second-order theory over trees [EJ91, JW96]. Further, it is known that algorithmic verification of μ-calculus properties of these abstractions is decidable (EXPTIME-complete) [Wal01, BS99]. From a practical standpoint, iterative computation of fixpoints, for instance in the above reachability requirement, naturally suggests symbolic evaluation, and symbolic model checkers such as SMV check CTL properties of finite-state models by compiling them into μ-calculus formulas [BCD$^+$92, McM93].

While designing a μ-calculus for nested trees, we have three goals in mind. First, we want to express the sort of context-sensitive requirements that the classical μ-calculus cannot express, while keeping model checking decidable. Second, we note that while a μ-calculus formula describes a terminating, iterative computation in finite-state programs, it does not necessarily do so in context-sensitive ones. Consequently, the μ-calculus cannot quite serve as a directive for computation of interprocedural requirements in pushdown models. While designing our logic, we aim to make it an assembly language for interprocedural computations. In other words, a formula in it should describe a concrete computation such as the classic summarization-based algorithms [RHS95, BEM97, ABE$^+$05] for interprocedural slicing or data flow analysis. Third, the logic, ideally, should not be ad hoc and should be a canonical logic for regular requirements on nested trees.

Our candidate for such a logic is called NT-μ [ACM06a]. The variables of this fixpoint calculus evaluate not over sets of states, but rather over sets of substructures that capture *summaries* of computations in the "current" program block. The fixpoint operators in the logic then compute fixpoints of summaries. For a node s of a nested tree representing a call, consider the tree rooted at s such that the leaves correspond to exits from the current context. Each formula of the logic is evaluated over such a structure. The central construct of the logic corresponds to concatenation of substructures: for example, a formula $\langle call \rangle \varphi\{\psi\}$ asserts a constraint φ on the new context, and requires ψ to hold at a designated set of return points of this context (this designation is done by marking the leaves of a subtree using a set of *colors*—see Chapter 3 for more details). To state local reachability, we would ask, using the formula φ, that control returns to the current context, and, using ψ, that the local reachability property holds at some return point. While this requirement seems self-referential, it may be captured using a fixpoint formula.

We show that these features allow our calculus to express the context-sensitive program requirements that we discussed earlier. As for model checking, we find that this problem for NT-μ is not only decidable, but no more costly than traditional branching-time logics. The complexity of model checking is exponential in the formula and the context-sensitive abstraction. For a fixed formula, the problem is exponential in the number of control locations where a procedure in the abstraction may return and polynomial in the number of remaining locations. The precise general-case complexity is EXPTIME-complete, which matches that of model-checking the μ-calculus on these abstractions (and, for that matter, that for the far weaker logic CTL [Wal01]). The model checking algorithm works by computing fixpoints of the summary sets inductively, and illustrates how the semantics of the logic naturally suggests a model checking algorithm. Fulfilling our second design criterion, the kind of summary computation traditionally known in interprocedural slicing or data flow analysis is a special case of this algorithm. so that, just like the μ-calculus in case of finite-state programs, NT-μ can arguably be used as a language into which interprocedural fixpoint computations can be compiled.

Finally, we show that NT-μ satisfies our third design requirement: there are good reasons why it can be called a canonical logic for regular nested tree properties. The reason is that it has a deep relationship with *automata on (infinite) nested trees*, which we discuss next.

1.3.3 Automata on nested trees

Automata on infinite nested trees are a natural generalization of automata on infinite trees. The latter, which accept ω-regular tree languages, are known to have a rich theory and have found many applications in the last thirty years. While reading a node in a tree, a tree automaton can nondeterministically pick different combinations of states to be passed along tree edges. On the other hand, an automaton on nested trees can send states along tree edges *and jump edges*, so that its state while reading a node depends on the states at its parent and the jump-predecessor (if one exists). Since there is an explicit jump-edge from a call to its matching return in a nested tree, these automata are naturally capable of matching calls with returns.

Like tree automata, automata on nested trees come in nondeterministic and alternating flavors, and can accept nested trees by a variety of acceptance conditions—e.g., Büchi, parity, final state, etc. As parity is the most powerful of acceptance conditions commonly used in ω-automata theory, we mainly focus on two classes of such automata: nondeterministic parity automata on nested trees (NP-NTAs), and *alternating* parity automata on nested trees (AP-NTAs), though we also briefly discuss automata accepting by infinite trees by a *final state*. NP-NTAs can nondeterministically label a nested tree with states while maintaining constraints like "if a node is labeled q, then all its tree-children are labeled q_1 and all its jump-children are labeled q_2." They can express a variety of context-sensitive requirements, and are, pleasantly, also closed under intersection. However, they are not closed under complementation, so that they are unlikely to have an attractive logical characterization. However, AP-NTAs do not suffer from this drawback. These automata are closed under all Boolean operations (though not under projection) and are strictly more expressive than NP-NTAs. While they have an undecidable emptiness problem, their model-checking problem is EXPTIME-complete, matching that for alternating *tree* automata on context-sensitive abstractions.

In a result analogous to the equivalence between the μ-calculus and parity tree automata, we find that NT-μ has the same expressive power as AP-NTAs. This strengthens our belief that NT-μ is not just another fixpoint logic, but captures the essence of regularity in nested trees. Our proof offers polynomial translations from AP-NTAs to NT-μ and vice versa, as well as insights about the connection between runs of AP-NTAs and the notion of summaries in NT-μ. This result is especially intriguing as the model checking algorithms for NT-μ and AP-NTAs are very different in flavor—while the latter reduces to pushdown games, the former seems to have no connection to the various previously known results about trees, context-free languages, and pushdown graphs. It also helps us compare the expressiveness of NT-μ with that of classical temporal logics and the temporal logic CARET [AEM04], which is a linear-time temporal logic for context-sensitive specification. Finally, this also establishes that the satisfiability problem for NT-μ is *undecidable*. Note that this is not an issue as we are really only interested in the model checking problem; in fact the result serves to illustrate how powerful the logic NT-μ is.

1.3.4 Monadic second-order logic on nested trees

Given the appealing trinity of automata, μ-calculus, and monadic second-order logic for regular tree languages, we study *monadic second-order (MSO) logic over nested trees*, which has a predicate $(x \hookrightarrow y)$ that can check the existence of a jump-edge between two nodes. We show that MSO-logic is strictly more expressive than NP-NTAs, and that the matching predicate is too powerful leading to undecidable satisfiability and model checking problems. The undecidability proof shows that the difficulty lies with combining second-order existential quantification with the matching predicate. On the other hand, there seems to be no way to encode sets of summaries in MSO-logic over nested trees, and we conjecture that NT-μ cannot be translated to MSO-logic.

1.3.5 The Pal instrumentation language

Next we introduce Pal, a specification language based on the theory of context-sensitive specifications that can be used in automatic safety analysis of C programs. Since we are only interested in safety properties in this case, we do not need the full power of NT-μ or nested tree automata. It suffices to consider deterministic automata on finite *nested words*, which can be viewed as nested trees with a single tree path.

The application is as follows. In program analysis tools like Blast [HJM+02] and Slam [BR01], as in classical automata-theoretic model checking [Hol97], a specification is a word automaton (or monitor) with finite-state control-flow that accepts all "unsafe" program executions. Typical analysis constructs the "product" of a program and a monitor, in effect *instrumenting* the program with extra instructions, so that the input program fails its specification iff the product program fails an assertion. The latter is then checked for possible assertion failures. Monitors also find use in testing and runtime verification, where the product program is tested for assertion violations at runtime.

As the control-flow of such traditional monitors is given by finite word automata, they cannot express context-sensitive specifications. For example, the safety property "There are no two successive file-write operations without a file-read operation in between" can be expressed using a finite word automaton. However, the context-sensitive analog "There are no two successive file-write operations *in the same context* without a file-read operation in between" is not. Given the theory we have developed, this problem can be overcome if a program execution is modeled as a nested word rather than a word, and monitors are based on automata on nested words.

Accordingly, we present a specification language—called PAL—based on nested word automata, and a tool to instrument C code using it. Monitors in this language have a set of variables that encode the automaton's state; transitions are blocks of C code that alter the values of these variables. We permit the variables to have arbitrary data types—however, on data abstraction of a PAL monitor, we get a nested word automaton. This language extends the BLAST specification language [BCH+04], and while its richer foundations lets it state context-sensitive properties, it has syntax close to BLAST's and allows easy instrumentation. As is to be expected, the original program satisfies the context-sensitive requirement if and only if the instrumented program does not fail an assertion. The instrumented program may now be checked for assertion failures using a variety of static and dynamic methods, including model checking as well as static analysis, run-time verification, and testing.

Of course, being based on the theory of nested specifications, monitors in PAL work irrespective of whether recursion is present. While they are theoretically only as expressive as monitors in BLAST in the absence of recursion, they are more modular, succinct and comprehensible even in this case. The reason is that PAL makes it possible to break a program execution into parts wholly contained within a procedural context, and reason about these parts separately. In fact, PAL is the first automata-based specification language that is directly usable in model checking, but is designed with the goal of expressing interface contracts between procedures. We believe, therefore, that these monitors present an example of *structured specifications*, suitable for structured programs.

We have implemented PAL on top of the BLAST query language. We use this implementation to write context-sensitive requirements for a few real (or at least realistic) C programs. We show that a PAL monitor could be used to prevent a reported bug in `fcron`, a periodic command scheduler for Linux. We also show how to write Pal monitors expressing stack-sensitive security requirements for C code and enforcing context-sensitive error logging policies.

1.4 Context-sensitive reachability analysis

In Part II of this thesis, we study algorithms for *reachability analysis of context-sensitive program abstractions*. Our abstractions of choice are *recursive state machines* [ABE+05], or finite-state machines that can call other finite-state machines recursively. These machines (called *RSMs* from now on) are equivalent to pushdown systems, or finite-state machines equipped with stacks. They are also natural abstractions of recursive programs: each component finite-state machine models control flow within a procedure, and procedure calls and returns are modeled by calls and returns to/from other machines. Sound analysis of a program then involves algorithmic analysis of an RSM abstracting it.

In this thesis, we study the most basic and widely applicable form that such analysis takes: determination of reachability between states. Can an RSM, in some execution, start at a state v and reach the state v'? Because RSMs are pushdown models, any path that the RSM can take respects the nested structure of calls and returns, and reachability analysis of an RSM abstraction of a program gives a *context-sensitive program analysis*. A classic application is interprocedural data-flow analysis— "can a data-flow fact reach a certain program point along a path respecting the nesting of procedure calls?" In fact, the problem has a reduction from the problem of model checking safety properties on context-sensitive program abstractions. Consider a safety property: "the program never reaches an error state." To prove that the abstraction does *not* satisfy this property, we need to determine reachability in it. As a result, algorithms solving this problem are at the heart of successful software analysis tools like Slam [BR01]. The problem also shows up in many other program analysis contexts—for example field-sensitive alias analysis [Rep98], type-based flow analysis [RF01], and shape analysis [Rep98].

Reachability for RSMs is equivalent to a well-known graph problem called *context-free language (CFL) reachability*. The question here is: given an edge-labeled directed graph and a context-free grammar over the edge labels, is there a path from node s to node t in the graph that is labeled by a word generated by the grammar? This problem, which may be viewed as a generalization of context-free recognition, was originally phrased in the context of database theory [Yan90], where it was shown that Datalog chain query evaluation on the graph representation of a database is equivalent to single-source, single-sink CFL-reachability. It has since been identified as a central problem in program analysis [Rep98, MR00].

All known algorithms for RSM and CFL-reachability follow a dynamic-programming scheme known in the literature as *summarization* [SP81, ABE$^+$05, BEM97]. The idea here is to derive reachability facts of the form (v, v'), which says that the RSM can start at state v with an empty stack and end at state v' with an empty stack. The most well-known algorithms following this scheme [HRS95, RHS95] discover such pairs enumeratively via graph traversal. Unlike context-free recognition, which has a well-known subcubic solution [Val75], RSM and CFL-reachability have not been known to have subcubic algorithms even in the single-sink, single-source case (for RSM-reachability, the size of an instance is the number of states in it; for CFL-reachability, it is the number of nodes in the input graph). This raises the question: are these problems intrinsically cubic? The question is especially interesting in program analysis as problems like interprocedural data-flow analysis and slicing are not only solvable using RSM-reachability, but also provably as hard. Believing that the answer is "yes", researchers have sometimes attributed the "cubic bottleneck" of these problems to the hardness of RSM or CFL-reachability [Rep98, MR00].

In this thesis, we observe that summarization can benefit from a known technique [Ryt83, Ryt85] for speeding up certain kinds of dynamic programming. The idea, developed in the context of language recognition for two-way pushdown automata, is to represent a computation accessing a table as a computation on row and column sets, which are stored using a “fast” set data structure. The latter, a standard data structure in the algorithms literature [ADKz70, Cha07], splits each operation involving a pair of sets into a series of operations on pairs of sets drawn from a small sub-universe. If the sub-universes are sufficiently small, all queries on them may be looked up from a table precomputed exhaustively, allowing us to save work during an expensive main loop. When transferred to the RSM-reachability problem with slight modifications, Rytter’s method leads to an algorithm that phrases the computation of reachability as a sequence of operations on sets of RSM states, and has an $O(n^3/\log n)$ time complexity. The technique may also be applied to the standard algorithm for CFL-reachability, referenced for example by Melski and Reps [MR00], leading to a similar speedup. This implies subcubic solutions for Datalog chain query evaluation as well as the many program analysis applications of RSM-reachability.

Our other contribution is an observation that the reachability problem for RSMs gets easier, so far as worst-case complexity is concerned, as recursion is restricted. We study the reachability problem for *bounded-stack recursive state machines*, which are RSMs where the stack never grows unboundedly in any execution. Machines of this sort have a clear interpretation in program analysis: they capture the flow of control in procedural programs without infinite recursive loops. In spite of this extra structure, they have not been known to have faster reachability algorithms than general RSMs (note that a bounded-stack RSM is in fact a finite-state machine—however, the latter can be exponentially larger than the RSM, so that it is not an option to analyze it instead of applying an RSM-reachability algorithm). We show that it is possible to exploit this structure during reachability analysis. The key observation is that empty-stack-to-empty-stack reachability facts in bounded-stack RSMs can be derived in a *depth-first order*—i.e., if state u has an edge to state v, it is possible to first infer all the states empty-stack-to-empty-stack reachable from

v and then use this information to infer the states reachable this way from v (this is not possible for general RSMs). It turns out that, as a result, we can solve the reachability problem using a transitive closure algorithm for directed graphs that allows the following kind of modifications to the instance: "for an edge (u, v) that goes from one strongly connected component to another, compute all descendants v' of v and add some edges from u based on the answer." Unfortunately, none of the existing subcubic algorithms for transitive closure can handle such modifications. Consequently, we derive a new transitive closure algorithm for directed graphs that can.

Our transitive closure algorithm speeds up a procedure based on Tarjan's algorithm to determine the strongly connected components of a graph. Such algorithms have a sizable literature [Pur70, EKS77, Sch83]. Their attraction in our setting is that they perform one depth-first traversal of the input graph, computing closure using set operations along the way, so that it is possible to weave the treatment of added edges into the discovery of edges in the original graph. The idea behind the speedup is, once again, to reuse computations on small patterns common to set computations, except this time, it can be taken further and yields a complexity of $O(\min\{mn/\log n, n^3/\log^2 n\})$, where n is the number of nodes in the graph and m the number of edges. This directly leads to an $O(n^3/\log^2 n)$ solution for all-pairs reachability in bounded-stack RSMs.

We finish our study of the interplay of recursion and reachability in RSMs with a note on the reachability problem for *hierarchical state machines* [AY98]. These machines can model control flow in structured programs without recursive calls and form a proper subclass of bounded-stack RSMs. The one published reachability algorithm for such models is cubic [AY98]; here, we give a simple alternative that has the same complexity as boolean matrix multiplication. While this algorithm is almost trivial, taken together with our other results, it indicates a gradation in the complexity of RSM-reachability as recursion is constrained.

1.5 Summary of contributions

We can now summarize the contributions of this thesis. First, we have phrased the branching-time model checking problem for pushdown abstractions in a new way, using nested trees rather than tree unfoldings. Second, we have introduced logics and finite automata that operate on nested trees rather than trees, and have found that they can express a variety of interesting non-regular specifications such as pre/post conditions, interprocedural data flow specifications, and security requirements involving the stack. Third, we have shown that these logics and automata have a robust theory, and that their model checking problem is not only decidable (as opposed to that of pushdown specifications), but no more costly than their weaker counterparts for tree models. Fourth, we have implemented a "context-sensitive" software monitoring language called PAL that applies the above theory in the special case of safety properties. Fifth, we have given the first subcubic algorithm for the reachability problem of recursive state machines (or, equivalently, pushdown systems or CFL-reachability), the central problem of interprocedural software analysis that was for long believed to be intrinsically cubic. We have also identified a better algorithm for the problem if recursion is restricted.

1.6 Thesis outline

The thesis is divided into two parts.

Part I, which studies context-sensitive requirement specification, is arranged as follows. In Chapter 2, we define nested trees and nested state machines. In Chapter 3, we develop the fixpoint calculus NT-μ. In Chapter 4, we introduce nondeterministic and alternating automata over nested trees and prove the equivalence of NT-μ and bisimulation-closed NTAs. We also study the logic MSO_{mr}. Chapter 5 introduces the PAL software specification language and discusses its implementation and a few case studies.

Part II, which studies algorithms for reachability analysis of context-sensitive program abstractions, has only one chapter: Chapter 6.

We wrap up the thesis with some discussion in Chapter 7.

Part I

Logics

Chapter 2

Preliminaries

In this chapter, we define *nested trees*, the models for the logics and automata we use for context-sensitive specification, and *nested state machines*, the program abstractions on which we perform model checking.

2.1 Nested trees

In the formal methods literature, it is common to model the branching behavior of a nondeterministic program using an infinite tree [CGP99] on which specification logics and automata are interpreted. The nondeterminism in the program is modeled via tree branching, so that each possible program execution is a path in the tree. However, this tree does not capture the nesting of procedure calls and returns in a program. At any given time point during the execution, the tree edge relation tells us what the possible program states are at the next time point. It does not, however, give us the program states that can possibly arise when a procedure call made at the current point returns. As a result, a tree model of a program's behavior is not suitable for context-sensitive specifications. As a replacement, we propose a new structure called *nested tree*, obtained by augmenting a tree capturing a structured program's behavior with an extra edge relation, known as the *jump-edge relation*.

Intuitively, a jump-edge connects a tree node representing a procedure call to the node representing the matching return. Thus, this definition inherits a bit more of the structure of the program than a tree model would. As calls and returns in structured programs are nested, jump-edges in nested trees do not cross, and calls and returns are defined respectively as sources and targets of jump-edges. In addition, since a procedure call may not return along all possible program paths, a call-node s may have jump-successors along some, but not all, paths from it. If this is the case, we add a jump-edge from s to a special node ∞.

Formally, let $T = (S, r, \to)$ be an unordered infinite tree with node set S, root r and edge relation $\to \subseteq S \times S$. Let $\xrightarrow{+}$ denote the transitive (but not reflexive) closure of the edge relation, and let a (finite or infinite) *path* in T from node s_1 be a (finite or infinite) sequence $\pi = s_1 s_2 \ldots s_n \ldots$ over S, where $n \geq 2$ and $s_i \to s_{i+1}$ for all $1 \leq i$.
A *nested tree* is a directed acyclic graph $(T, \hookrightarrow)$, where $\hookrightarrow \subseteq S \times (S \cup \infty)$ is a set of jump-edges. A node s such that $s \hookrightarrow t$ or $s \hookrightarrow \infty$ (similarly $t \hookrightarrow s$) for some t is a *call* (*return*) node; the remaining nodes are said to be *local*. The intuition is that if $s \hookrightarrow t$, then a call at s returns at t; if $s \hookrightarrow \infty$, then there exists a path from s along which the call at s never returns. We note that the sets of call, return and local nodes are disjoint. The jump-edges must satisfy:

1. if $s \hookrightarrow t$, then $s \xrightarrow{+} t$, and we do not have $s \to t$ (in other words, jump-edges represent non-trivial forward jumps);

2. if $s \hookrightarrow t$ and $s \hookrightarrow t'$, then neither $t \xrightarrow{+} t'$ nor $t' \xrightarrow{+} t$ (this captures the intuition that a call-node has at most one matching return along every path from it);

3. if $s \hookrightarrow t$ and $s' \hookrightarrow t$, then $s = s'$ (every return node has a unique matching call);

4. for every call node s, we have either (a) on every path from s, there is a node t such that $s \hookrightarrow t$, or (b) $s \hookrightarrow \infty$ (a call either returns along all paths, or does

not);

5. if there is a path π such that for nodes s, t, s', t' lying on π we have $s \xrightarrow{+} s'$, $s \hookrightarrow t$, and $s' \hookrightarrow t'$, then either $t \xrightarrow{+} s'$ or $t' \xrightarrow{+} t$ (jump-edges along a path do not cross);

6. for every pair of call-nodes s, s' on a path π such that $s \xrightarrow{+} s'$, if there is no node t on π such that $s' \hookrightarrow t$, then a node t' on π can satisfy $s \hookrightarrow t'$ only if $t' \xrightarrow{+} s'$ (if a call does not return, neither do the calls pending when it was invoked).

A *nested word* is a nested tree as above, except the structure T is a word.

For an alphabet Σ, a Σ-labeled nested tree is a structure $\mathcal{T} = (T, \hookrightarrow, \lambda)$, where $(T, \hookrightarrow)$ is a nested tree with node set S, and $\lambda : S \to \Sigma$ is a node-labeling function (labeled nested words are defined in the same way). All nested trees in this paper are Σ-labeled.

```
input x;

procedure foo()
{
L1:  write(e);
     if(x)
L2:    foo();
     else
L3:    think;
     while (x)
L4:    read(e);
L5:  return;
}
```

Figure 2.1: A sample program

Consider the recursive procedure *foo* in Fig. 2.1. The procedure may read or write an expression e or perform an action *think*, has branching dependent on an input variable x, and can call itself recursively. Actions of the program are marked by labels L1–L5 for easy reference. We will abstract this program and its behaviors, and subsequently specify it using temporal logics and automata.

Figure 2.2 shows a part of a nested tree modeling the branching behavior of this program. Because the loop and the branch in the procedure depend on an environment-dependent variable, we model them by a nondeterministic loop and a nondeterministic branch. The choice of the alphabet Σ labeling this tree depends on the desired level of detail. We choose it to consist of subsets of a set of atomic propositions AP, comprising the propositions *wr*, *rd*, *en*, *ex*, *tk*, and *end*, respectively encoding a *write* statement, a *read* statement, a procedure call leading to a beginning of a new context, the return point once a context ends, the statement *think*, and the statement *return*. A node is labeled by the proposition for a statement if it is the control point *from which* the statement is executed— e.g., the control point immediately preceding a read statement is labeled *rd*. Each path in the underlying tree captures a sequence of program statements—for example, the path fragment starting at the node s and ending at s' captures a (partial) execution that first executes a *write*, then calls *foo* recursively, then writes again, then makes another recursive call, ending once it has exited both calls. Note that some of the maximal paths are finite—these capture terminating executions of the program—and some are not. Note in particular how a call may return along some paths from it, and yet not on some others. A path in the nested tree that takes a jump-edge whenever possible is interpreted as a local path through the top-level context.

If $s \hookrightarrow t$, then s is the jump-predecessor of t and t the jump-successor of s. Edges from a call node and to a return node are known as *call* and *return* edges; the remaining edges are *local*. The fact that an edge (s, t) exists and is a call, return or local edge is denoted by $s \xrightarrow{call} t$, $s \xrightarrow{ret} t$, or $s \xrightarrow{loc} t$. For a nested tree $\mathcal{T} = (T, \hookrightarrow, \lambda)$ with edge set E, the *tagged tree* of $\mathcal{T}$ is the node and edge-labeled tree $Struct(\mathcal{T}) = (T, \lambda, \eta : E \to \{call, ret, loc\})$, where $\eta(s, t) = a$ iff $s \xrightarrow{a} t$.

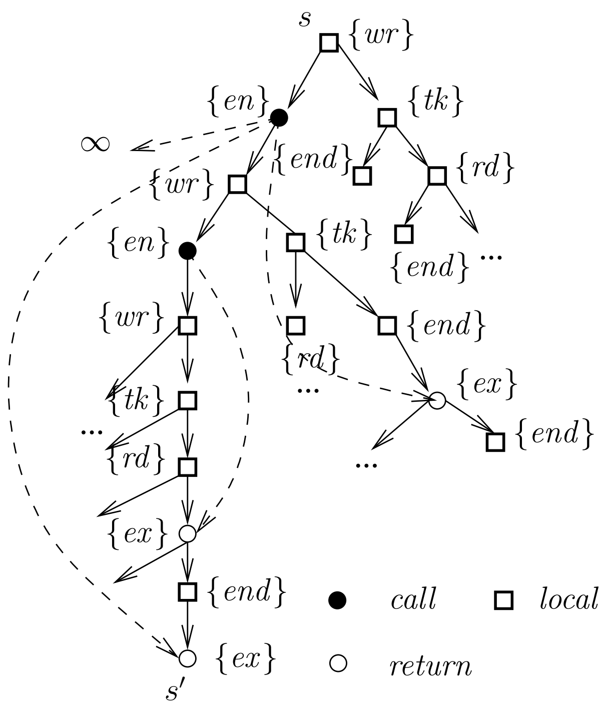

Figure 2.2: A nested tree

A few observations: first, the sets of call, return and local edges define a partition of the set of tree edges. Second, if $s \xrightarrow{ret} s_1$ and $s \xrightarrow{ret} s_2$ for distinct s_1 and s_2, then s_1 and s_2 have the same jump-predecessor. Third, the jump-edges in a nested tree are completely captured by the edge labeling in the corresponding tagged tree, so that we can *reconstruct* a nested tree $\mathcal{T}$ from $Struct(\mathcal{T})$. We show how by making a stronger statement. Consider *any* tree $T = (S, r, \xrightarrow{)}$ whose edges are labeled by tags *call*, *ret* and *loc* and that satisfies the constraint: if a node has an outgoing edge labeled *call*, then all its outgoing edges are labeled *call*. Note that tagged trees derived from nested trees satisfy this constraint. Now, let us call a word $\beta \in I^*$ *balanced* if it is of the form

$$\beta := \beta\beta \mid call.\beta.ret \mid loc.$$

Let us now define a relation $\hookrightarrow' \subseteq S \times S$ as: for all s, s', we have $s \hookrightarrow' t$ iff

1. there is a path $s_0 s_1 s_2 \ldots s_n$ such that $s_0 = s$ and $s_n = s'$ in $\mathit{Struct}(\mathcal{T})$;

2. the word $\eta(s_0, s_1).\eta(s_1, s_2) \ldots \eta(s_{n-1}, s_n)$ is balanced;

3. for no $i > 0$ is the word $\eta(s_i, s_{i+1}).\eta(s_{i+1}, s_{i+2}). \ldots \eta(s_{n-1}, s_n)$ balanced.

Now consider the set S_{uc} of nodes s_{uc} such that: (1) outgoing edges from s_{uc} are labeled *call*, and (2) there is no $t \in S$ such that $(s_{uc}, t) \in \hookrightarrow')$. Let us construct the relation $\hookrightarrow'' = \hookrightarrow' \cup \{(s_{uc}, \infty) : s_{uc} \in S_{uc}\}$. It is easily verified that $(T, \hookrightarrow'')$ is a nested tree, and that is $T = \mathit{Struct}(\mathcal{T})$ for some nested tree $\mathcal{T}$, then $\mathcal{T} = (T, \hookrightarrow'')$.

Let $NT(\Sigma)$ be the set of Σ-labeled nested trees. A language of nested trees is a subset of $NT(\Sigma)$.

2.1.1 Ordered, binary nested trees

Note that in the definition of nested trees we have given, the tree structure underlying a nested tree is unordered. While this is the definition we will use as the default definition in this thesis, we will find use for *ordered, binary nested trees* in a few occasions.

Let $T = (S, r, \rightarrow_1, \rightarrow_2)$ be an ordered binary tree, where S is a set of nodes, r is the root, and $\rightarrow_1, \rightarrow_2 \subseteq S \times S$ are the left and right edge relations. Then $(T, \hookrightarrow)$ is an ordered, binary nested tree if $((S, r, \rightarrow_1 \cup \rightarrow_2), \hookrightarrow)$ is a nested tree by our earlier definition. Labeled, ordered nested trees are analogous to labeled, unordered nested trees: for an alphabet Σ, a Σ-labeled ordered nested tree is a structure $\mathcal{T} = (T, \hookrightarrow, \lambda)$, where $(T, \hookrightarrow)$ is a nested tree with node set S, and $\lambda : S \rightarrow \Sigma$ is a node-labeling map.

2.2 Nested state machines

Now we define our program abstractions: *nested state machines* (NSMs). Like pushdown automata and recursive state machines [ABE+05], NSMs are suitable for precisely modeling changes to the program stack due to procedure calls and returns. The main difference is that the semantics of an NSM is defined using a nested tree rather than using a stack.

Syntax

Let AP be a fixed set of atomic propositions; let us fix $\Sigma = 2^{AP}$ as an alphabet of *observables*. A *nested state machine* (NSM) is a structure of the form $\mathcal{M} = \langle V_{loc}, V_{call}, V_{ret}, v_{in}, \kappa, \Delta_{loc}, \Delta_{call}, \Delta_{ret} \rangle$. Here, V_{loc} is a finite set of *local states*, V_{call} a finite set of *call states*, and V_{ret} a finite set of *return states*. We write $V = V_{loc} \cup V_{call} \cup V_{ret}$. The state $v_{in} \in V$ is the *initial state*, and the map $\kappa : V \to \Sigma$ labels each state with an observable. There are three transition relations: a *local transition relation* $\Delta_{loc} \subseteq (V_{loc} \cup V_{ret}) \times (V_{loc} \cup V_{call})$, a *call transition relation* $\Delta_{call} \subseteq V_{call} \times (V_{loc} \cup V_{call})$, and a *return transition relation* $\Delta_{ret} \subseteq (V_{loc} \cup V_{ret}) \times V_{call} \times V_{ret}$.

A transition is said to be *from the state* v if it is of the form (v, v') or (v, v', v''), for some $v', v'' \in V$. If $(v, v') \in \Delta_{loc}$ for some $v, v' \in V$, then we write $v \xrightarrow{loc} v'$; if $(v, v') \in \Delta_{call}$, we write $v \xrightarrow{call} v'$; if $(v, v', v'') \in \Delta_{ret}$, we write $(v, v') \xrightarrow{ret} v''$. Intuitively, while modeling a program by an NSM, a call state models a program state from which a procedure call is performed; the call itself is modeled by a call transition in Δ_{call}. A return state of an NSM models a state to which the control returns once a called procedure terminates. The shift of control to a return state is modeled by a return transition (v, v', v'') in Δ_{ret}. Here, the states v and v'' are respectively the current and target states, and v' is the state from which the last "unmatched" call-move was made. The intuition is that when the NSM made a call transition from v', it pushed the state v' on an implicit stack. On return, v' is on top of the stack right before the return-move, which can depend on this state and, on completion, pops it off the stack. This captures the ability of a structured program to use its procedural stack, which is the essence of context-sensitivity. A state that is neither a call nor a return is a local state, and a transition that does not modify the program stack is a local transition.

Let us now abstract our example program into a nested state machine $\mathcal{M}_{foo}$. The abstraction simply captures control flow in the program, and consequently, has states v_1, v_2, v_3, v_4, and v_5 corresponding to lines L1, L2, L3, L4, and L5. We also have a state v_2' to which control returns after the call at L2 is completed. The set V_{loc} of local states is $\{v_1, v_3, v_4, v_5\}$, the single call state is v_2, and the single return state is v_2'. The initial state is v_1. Now, let us have propositions *rd*, *wr*, *tk*, *en*, *ex*, and *end* that hold respectively iff the current state represents the control point immediately before a read, write, think statement, procedure call, return point after a call, and return instruction. More precisely, $\kappa(v_1) = \{wr\}$, $\kappa(v_2) = \{en\}$, $\kappa(v_2') = \{ex\}$, $\kappa(v_3) = \{tk\}$, $\kappa(v_4) = \{rd\}$, and $\kappa(v_5) = \{end\}$ (for easier reading, we will, from now on, abbreviate singletons such as $\{rd\}$ just as rd).

The transition relations of $\mathcal{M}_{foo}$ are given by:

- $\Delta_{call} = \{(v_2, v_1)\}$
- $\Delta_{loc} = \{(v_1, v_2), (v_1, v_3), (v_2', v_4), (v_2', v_5), (v_3, v_4), (v_3, v_5), (v_4, v_4), (v_4, v_5)\}$, and
- $\Delta_{ret} = \{(v_5, v_2, v_2')\}$.

Branching-time semantics

The branching-time semantics of $\mathcal{M}$ is defined via a 2^{AP}-labeled unordered nested tree $\mathcal{T}(\mathcal{M})$, known as the *unfolding* of $\mathcal{M}$. Consider the V-labeled (unordered) nested tree $\mathcal{T}^V(\mathcal{M}) = (T, \hookrightarrow, \lambda)$, known as the *execution tree*, that is the unique nested tree satisfying the following conditions:

1. if r is the root of T, then $\lambda(r) = v_{in}$;

2. for every node s and every distinct call, return or local transition in $\mathcal{M}$ from $\lambda(s)$, s has precisely one outgoing call, return or local tree edge;

3. for every pair of nodes s and t, if $s \xrightarrow{a} t$, for $a \in \{\mathit{call}, \mathit{loc}\}$, in the tagged tree of this nested tree, then we have $\lambda(s) \xrightarrow{a} \lambda(t)$ in $\mathcal{M}$;

4. for every s, t, if $s \xrightarrow{\mathit{ret}} t$ in the tagged tree, then there is a node t' such that $t' \hookrightarrow t$ and $(\lambda(s), \lambda(t')) \xrightarrow{\mathit{ret}} \lambda(t)$ in $\mathcal{M}$.

Note that a node s is a call or return node in this nested tree respectively iff $\lambda(s)$ is a call and return state of $\mathcal{M}$. Now we have $\mathcal{T}(\mathcal{M}) = (T, \hookrightarrow, \lambda')$, where $\lambda'(s) = \kappa(\lambda(s))$ for all nodes s. For example, the nested tree in Figure 2.2 is the unfolding of $\mathcal{M}_{\mathit{foo}}$. While unfoldings of nested state machines are most naturally viewed as unordered nested trees, we can also define an NSM's unfolding as an ordered, binary nested tree. In this case, we fix an order on the transitions out of a state and allow at most two outgoing transitions from every state (we can expand the state set to make this possible). The left and right edge relations in the unfolding $\mathcal{T}_{\mathit{ord}}(\mathcal{M})$ respectively correspond to the 1st and 2nd transitions out of a state. We leave out the detailed definition.

Chapter 3

NT-μ: a fixpoint calculus for nested trees

In this chapter, we develop NT-μ, our modal fixpoint calculus interpreted on nested trees. The variables of this logic are evaluated not over sets of states, but over sets of subtrees that capture *summaries* of computations capturing procedural context. The fixpoint operators in the logic then compute fixpoints of summaries. The main technical result is that the logic NT-μ can be model-checked effectively on nested state machine abstractions of software.

The chapter is organized as follows. Section 3.1 introduces summaries, which are the models for logic NT-μ, and Section 3.2 defines the logic itself. In Section 3.3 we present context-sensitive requirements for software that can be expressed using NT-μ. Section 3.4 shows how nested state machine models of programs can be model-checked against NT-μ formulas, and Section 3.5 contains a result on expressiveness.

3.1 Summaries

Now we define *summaries*, the objects on which our logic is interpreted. These may be viewed as substructures of nested trees capturing procedural contexts; a summary models the branching behavior of a program from a state s to each return point of its context. Also, to capture different temporal obligations to be met on exiting via different exits, we introduce a *coloring* of these exits—intuitively, an exit gets color i if it is to satisfy the i-th requirement.

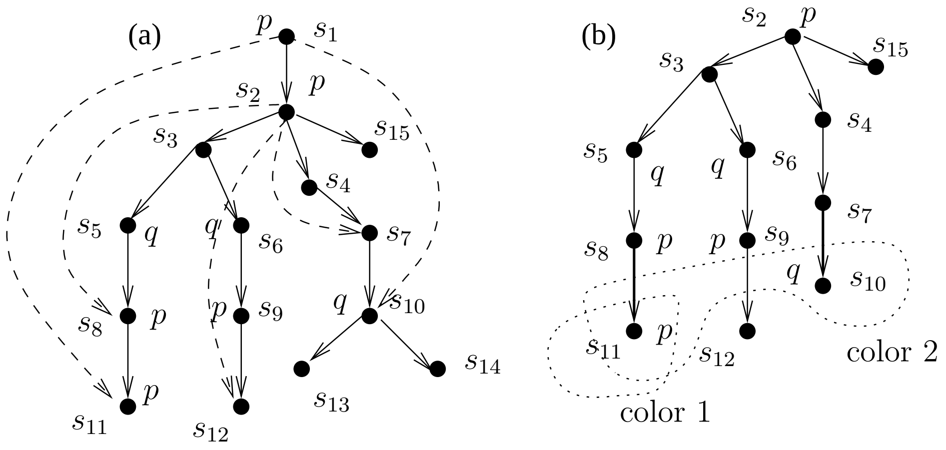

Figure 3.1: (a) A nested tree (b) A 2-colored summary

Formally, let a node t of $\mathcal{T}$ be called a *matching exit* of a node s if there is an s' such that $s' \xrightarrow{+} s$ and $s' \hookrightarrow t$, and there are no s'', t'' such that $s' \xrightarrow{+} s'' \xrightarrow{+} s \xrightarrow{+} t''$, and $s'' \hookrightarrow t''$. Intuitively, a matching exit of s is the first "unmatched" return along some path from s—for instance, in Figure 3.1-(a), the node s_8 and s_{12} are the matching exits of the node s_3, and s_{11} and s_{10} are the matching exits of s_2. Let the set of matching exits of s be denoted by $ME(s)$.

For a non-negative integer k, a *summary* $\mathbf{s}$ in $\mathcal{T}$ is a tuple $\langle s, U_1, U_2, \ldots, U_k \rangle$, where s is a node, $k \geq 0$, and $U_1, U_2, \ldots, U_k \subseteq ME(s)$ (such a summary is said to be *rooted* at s). For example, in the nested tree in Figure 3.1-(a), $\langle s_1 \rangle$ is a valid 0-colored summary, and $\langle s_2, \{s_{11}\}, \{s_{10}, s_{11}\} \rangle$ and $\langle s_3, \{s_8\}, \emptyset \rangle$ are valid 2-colored summaries. The set of summaries in a nested tree $\mathcal{T}$, each k-colored for some k, is denoted by $\mathbb{S}$. Note that such colored summaries are defined for all s, not just "entry" nodes of procedures.

Observe how each summary describes a subtree along with a coloring of some of its leaves. For instance, the summary $\mathbf{s} = \langle s_2, \{s_{11}\}, \{s_{10}, s_{11}\}\rangle$ marks the subtree in Figure 3.1-(b). Such a tree may be constructed by taking the subtree of $\mathcal{T}$ rooted at node s_2, and *chopping off* the subtrees rooted at $ME(s_2)$. Note that because of unmatched infinite paths from the root, such a tree may in general be infinite. Now, nodes s_{11} is assigned the color 1, and nodes s_{10} and s_{11} are colored 2. Note that the same matching exit might get multiple colors.

It is useful to contrast our definition of summaries with the corresponding definition for the linear-time setting. In this case, a pair (s, s'), where $s' \in ME(s)$, would suffice as a summary— in fact, this is the way in which traditional summarization-based decision procedures have defined summaries. For branching-time reasoning, however, such a simple definition is not enough.

3.2 The logic

3.2.1 Syntax

In addition to being interpreted over summaries, the logic NT-μ differs from classical calculi like the modal μ-calculus [Koz83] in a crucial way: its syntax and semantics explicitly recognize the procedural structure of programs. This is done using modalities such as $\langle call\rangle$, $\langle ret\rangle$ and $\langle loc\rangle$ that can distinguish between call, return, and local edges in a nested tree. Also, an NT-μ formula can enforce different "return conditions" at differently colored returns in a summary by passing formulas as "parameters" to *call* modalities.

Formally, let AP be a finite set of atomic propositions, *Var* be a finite set of variables, and $\{R_1, R_2, \ldots\}$ be a countable, ordered set of *markers*. For $p \in AP$, $X \in Var$, and $k \geq 0$, formulas φ of NT-μ are defined by:

$$\varphi := p \mid \neg p \mid X \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \mu X.\phi \mid \nu X.\phi \mid \langle call\rangle\ \varphi\{\psi_1, \psi_2, ..., \psi_k\} \mid$$

$$[call]\ \varphi\{\psi_1, \psi_2, ..., \psi_k\} \mid \langle loc\rangle\ \varphi \mid [loc]\ \varphi \mid \langle ret\rangle\ R_i \mid [ret]\ R_i,$$

where $k \geq 0$ and $i \geq 1$. Let us define the syntactic shorthands $tt = p \vee \neg p$ and $\mathit{ff} = p \wedge \neg p$ for some $p \in AP$. Also, let the *arity* of a NT-μ formula φ be the maximum k such that φ has a subformula of the form $\langle call\rangle\varphi'\{\psi_1, \ldots, \psi_k\}$ or $[call]\varphi'\{\psi_1, \ldots, \psi_k\}$.

Intuitively, the markers R_i in a formula are bound by $\langle call\rangle$ and $[call]$ modalities, and variables X are bound by fixpoint quantifiers μX and νX. We require our *call*-formulas to bind all the markers in their scope. Formally, let the *maximum marker index* $ind(\varphi)$ of a formula φ be defined inductively as:

$$\begin{aligned}
ind(\varphi_1 \vee \varphi_2) &= ind(\varphi_1 \wedge \varphi_2) = \max\{ind(\varphi_1), ind(\varphi_2)\} \\
ind(\langle loc\rangle\varphi) &= ind([loc]\varphi) = ind(\mu X.\varphi) = ind(\nu X.\varphi) \\
&= ind(\varphi) \\
ind(\langle ret\rangle R_i) &= ind([ret]R_i) = i \\
ind(p) &= ind(X) = 0 \text{ for } p \in AP,\ X \in Var \\
ind(\langle call\rangle\varphi\{\psi_1, \ldots, \psi_k\}) &= ind([call]\varphi\{\psi_1, \ldots, \psi_k\}) \\
&= \max\{ind(\psi_1), \ldots, ind(\psi_k)\}.
\end{aligned}$$

We are only interested in formulas where for every subformula $\langle call\rangle\chi'\{\psi_1, \ldots, \psi_k\}$ or $[call]\chi'\{\psi_1, \ldots, \psi_k\}$, we have $ind(\chi') \leq k$. Such a formula φ is said to be *marker-closed* if $ind(\varphi) = 0$.

The set $Free(\varphi)$ of free variables in a NT-μ formula φ is defined as:

$$\begin{aligned}
Free(\varphi_1 \vee \varphi_2) &= Free(\varphi_1 \wedge \varphi_2) = Free(\varphi_1) \cup Free(\varphi_2) \\
Free(\langle loc\rangle\varphi) &= Free([loc]\varphi) = Free(\varphi) \\
Free(\langle ret\rangle R_i) &= Free([ret]R_i) = \emptyset \\
Free(\langle call\rangle\varphi\{\psi_1, \ldots, \psi_k\}) &= Free([call]\varphi\{\psi_1, \ldots, \psi_k\}) = Free(\varphi) \cup (\bigcup_i^k Free(\psi_i)) \\
Free(p) &= Free(\neg p) = \emptyset \text{ for } p \in AP
\end{aligned}$$

$$Free(X) = \{X\} \text{ for } X \in Var$$
$$Free(\mu X.\varphi) = Free(\nu X.\varphi) = Free(\varphi) \setminus \{X\}.$$

A formula φ is said to be *variable-closed* if it has $Free(\varphi) = \emptyset$. We call φ *closed* if it is marker-closed and variable-closed.

3.2.2 Semantics

Like in the modal μ-calculus, formulas in NT-μ encode sets, in this case sets of summaries. Also like in the μ-calculus, modalities and boolean and fixed-point operators allow us to encode computations on these sets.

To understand the semantics of local ($\langle loc \rangle$ and $[loc]$) modalities in NT-μ, consider the 1-colored summary $\mathbf{s} = \langle s_3, \{s_8\} \rangle$ in the tree $\mathcal{T}$ in Figure 3.1-(a). We observe that when control moves from node s_3 to s_5 along a local edge, the current context stays the same, though the set of returns that can end it and are reachable from the current control point can get restricted — i.e., $ME(s_5) \subseteq ME(s_3)$. Consequently, the 1-colored summary $\mathbf{s}' = \langle s_5, \{s_8\} \rangle$ describes program flow from s_5 to the end of the current context, and is the *local successor* of the summary $\mathbf{s}$. NT-μ allows us to use modalities $\langle loc \rangle$ and $[loc]$ to assert requirements on such local successors. For instance, in this case, the summary $\mathbf{s}$ will be said to satisfy the formula $\langle loc \rangle q$, as $\mathbf{s}'$ satisfies q.

An interesting visual insight about the structure of the tree $\mathcal{T}_s$ for $\mathbf{s}$ comes from Figure 3.2-(a). Note that the tree $\mathcal{T}_{s'}$ for $\mathbf{s}'$ "hangs"' from the former by a local edge; additionally, (1) every leaf of $\mathcal{T}_{s'}$ is a leaf of $\mathcal{T}_s$, and (2) such a leaf gets the same color in $\mathbf{s}$ and $\mathbf{s}'$.

Succession along call edges is more complex, because along such an edge, a frame is pushed on a program's stack and a new procedural context gets defined. In

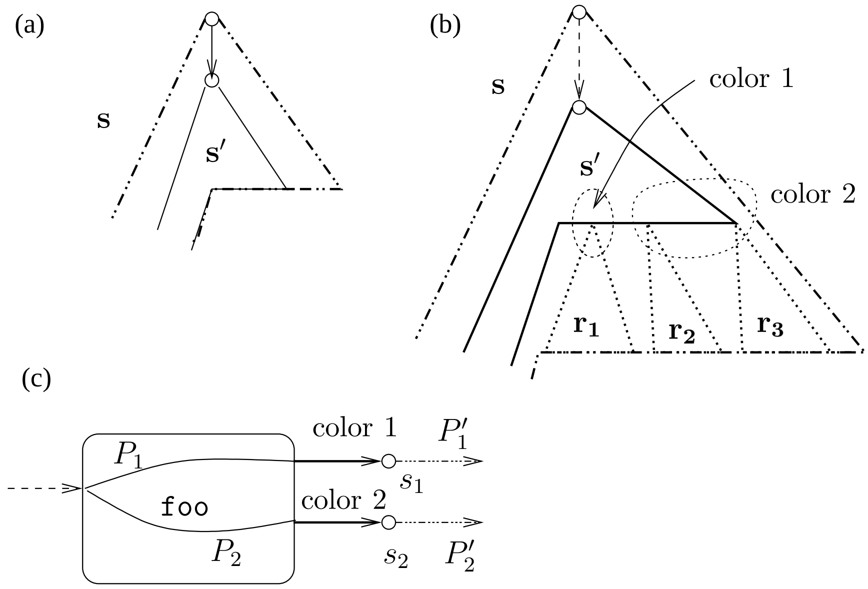

Figure 3.2: (a) Local modalities (b) Call modalities (c) Matching contexts.

Figure 3.1-(a), take the summary $\mathbf{s} = \langle s_1 \rangle$, and demand that it satisfies the two-parameter call formula $\langle call \rangle \varphi' \{q, p\}$. This formula asserts a condition on a subtree that: (1) is rooted at a child of s_1, and (2) has colors 1 and 2 assigned respectively to the leaves satisfying p and q. Clearly, a possible such summary is $\mathbf{s}' = \langle s_2, \{s_{10}\}, \{s_{11}\} \rangle$. Our formula requires that $\mathbf{s}'$ satisfies φ'. In general, we could have formulas of the form $\varphi = \langle call \rangle \varphi' \{\psi_1, \psi_2, \ldots, \psi_k\}$, where ψ_i are arbitrary NT-μ formulas.

Visually, we find that the above requires a split of the nested tree $\mathcal{T}_s$ for summary $\mathbf{s}$ in the way shown in Figure 3.2-(b). The root of this structure must have a *call*-edge to the root of the tree for $\mathbf{s}'$, which must satisfy φ. At each leaf of $\mathcal{T}_{s'}$ colored i, we must be able to *concatenate* a summary tree $\mathcal{T}_r$ satisfying ψ_i such that (1) every leaf in $\mathcal{T}_r$ is a leaf of $\mathcal{T}_s$, and (2) each such leaf gets the same set of colors in $\mathcal{T}_s$ and $\mathcal{T}_r$.

As for the return modalities, we use them to assert that we return at a node colored i. Because the binding of these colors to temporal requirements was fixed at a context that called the current context, the *ret*-modalities let us relate a path in the latter with the continuation of a path in the former. For instance, in Figure 3.2-(c), where the rectangle abstracts the part of a program unfolding within the body of a procedure `foo`, the marking of return points s_1 and s_2 by colors 1 and 2 is visible inside `foo` as well as at the call site of `foo`. This lets us match paths P_1 and P_2 inside `foo` respectively with paths P_1' and P_2' in the calling procedure. This lets NT-μ capture the pushdown structure of branching-time runs of a procedural program.

Let us now describe the semantics of NT-μ formally. A NT-μ formula φ is interpreted in an *environment* that interprets variables in $Free(\varphi)$ as sets of summaries in a nested tree $\mathcal{T}$ with node set S. Formally, an *environment* is a map $\mathcal{E} : Free(\varphi) \to 2^{\mathbb{S}}$. Let us write $[\![\varphi]\!]_{\mathcal{E}}^{\mathcal{T}}$ to denote the set of summaries in $\mathcal{T}$ satisfying φ in environment $\mathcal{E}$ (usually $\mathcal{T}$ will be understood from the context, and we will simply write $[\![\varphi]\!]_{\mathcal{E}}$). For a summary $\mathbf{s} = \langle s, U_1, U_2, \ldots, U_k \rangle$, where $s \in S$ and $U_i \subseteq ME(s)$ for all i, $\mathbf{s}$ satisfies φ, i.e., $\mathbf{s} \in [\![\varphi]\!]_{\mathcal{E}}$, if and only if one of the following holds:

- $\varphi = p \in AP$ and $p \in \lambda(s)$
- $\varphi = \neg p$ for some $p \in AP$, and $p \notin \lambda(s)$
- $\varphi = X$, and $\mathbf{s} \in \mathcal{E}(X)$
- $\varphi = \varphi_1 \vee \varphi_2$ such that $\mathbf{s} \in [\![\varphi_1]\!]_{\mathcal{E}}$ or $\mathbf{s} \in [\![\varphi_2]\!]_{\mathcal{E}}$
- $\varphi = \varphi_1 \wedge \varphi_2$ such that $\mathbf{s} \in [\![\varphi_1]\!]_{\mathcal{E}}$ and $\mathbf{s} \in [\![\varphi_2]\!]_{\mathcal{E}}$
- $\varphi = \langle call \rangle \varphi' \{\psi_1, \psi_2, ..., \psi_m\}$, and there is a $t \in S$ such that (1) $s \xrightarrow{call} t$, and (2) the summary $\mathbf{t} = \langle t, V_1, V_2, \ldots, V_m \rangle$, where for all $1 \leq i \leq m$, $V_i = ME(t) \cap \{s' : \langle s', U_1 \cap ME(s'), \ldots, U_k \cap ME(s') \rangle \in [\![\psi_i]\!]_{\mathcal{E}}\}$, is such that $\mathbf{t} \in [\![\varphi']\!]_{\mathcal{E}}$
- $\varphi = [call]\ \varphi' \{\psi_1, \psi_2, ..., \psi_m\}$, and for all $t \in S$ such that $s \xrightarrow{call} t$, the summary $\mathbf{t} = \langle t, V_1, V_2, \ldots, V_m \rangle$, where for all $1 \leq i \leq m$, $V_i = ME(t) \cap \{s' : \langle s', U_1 \cap ME(s'), \ldots, U_k \cap ME(s') \rangle \in [\![\psi_i]\!]_{\mathcal{E}}\}$, is such that $\mathbf{t} \in [\![\varphi']\!]_{\mathcal{E}}$

- $\varphi = \langle loc \rangle\ \varphi'$, and there is a $t \in S$ such that $s \xrightarrow{loc} t$ and the summary $\mathbf{t} = \langle t, V_1, V_2, \ldots, V_k \rangle$, where $V_i = ME(t) \cap U_i$, is such that $\mathbf{t} \in [\![\varphi']\!]_{\mathcal{E}}$

- $\varphi = [loc]\ \varphi'$, and for all $t \in S$ s.t. $s \xrightarrow{loc} t$, the summary $\mathbf{t} = \langle t, V_1, V_2, \ldots, V_k \rangle$, where $V_i = ME(t) \cap U_i$, is such that $\mathbf{t} \in [\![\varphi']\!]_{\mathcal{E}}$

- $\varphi = \langle ret \rangle\ R_i$, and there is a $t \in S$ such that $s \xrightarrow{ret} t$ and $t \in U_i$

- $\varphi = [ret]\ R_i$, and for all $t \in S$ such that $s \xrightarrow{ret} t$, we have $t \in U_i$

- $\varphi = \mu X.\varphi'$, and $\mathbf{s} \in \mathbf{S}$ for all $\mathbf{S} \subseteq \mathbb{S}$ satisfying $[\![\varphi']\!]_{\mathcal{E}[X:=\mathbf{S}]} \subseteq \mathbf{S}$

- $\varphi = \nu X.\varphi'$, and there is some $\mathbf{S} \subseteq \mathbb{S}$ such that (1) $\mathbf{S} \subseteq [\![\varphi']\!]_{\mathcal{E}[X:=\mathbf{S}]}$ and (2) $\mathbf{s} \in \mathbf{S}$.

Here $\mathcal{E}[X := \mathbf{S}]$ is the environment $\mathcal{E}'$ such that: (1) $\mathcal{E}'(X) = \mathbf{S}$, and (2) $\mathcal{E}'(Y) = \mathcal{E}(Y)$ for all variables $Y \neq X$. We say a node s *satisfies* a formula φ if the 0-colored summary $\langle s \rangle$ satisfies φ. A nested tree $\mathcal{T}$ rooted at s_0 is said satisfy φ if s_0 satisfies φ (we denote this by $\mathcal{T} \models \varphi$). The language of φ, denoted by $\mathcal{L}(\varphi)$, is the set of nested trees satisfying φ.

A few observations are in order. First, while NT-μ does not allow formulas of form $\neg\varphi$, it is closed under negation so long as we stick to closed formulas. Given a closed NT-μ formula φ, consider the formula $Neg(\varphi)$, defined inductively in the following way:

- $Neg(p) = \neg p$, $Neg(\neg p) = p$, $Neg(X) = X$

- $Neg(\varphi_1 \vee \varphi_2) = Neg(\varphi_1) \wedge Neg(\varphi_2)$, and $Neg(\varphi_1 \wedge \varphi_2) = Neg(\varphi_1) \vee Neg(\varphi_2)$

- If $\varphi = \langle call \rangle\ \varphi'\{\psi_1, \psi_2, ..., \psi_k\}$, then
$Neg(\varphi) = [call]\ Neg(\varphi')\{Neg(\psi_1), Neg(\psi_2), \ldots, Neg(\psi_k)\}$

- If $\varphi = [call]\ \varphi'\{\psi_1, \psi_2, ..., \psi_k\}$, then
$Neg(\varphi) = \langle call \rangle\ Neg(\varphi')\{Neg(\psi_1), Neg(\psi_2), \ldots, Neg(\psi_k)\}$

- $Neg(\langle loc\rangle\varphi') = [loc]Neg(\varphi')$, and $Neg([loc]\varphi') = \langle loc\rangle Neg(\varphi')$
- $Neg(\langle ret\rangle R_i) = [ret]R_i$, and $Neg([ret]R_i) = \langle ret\rangle R_i$
- $Neg(\mu X.\varphi) = \nu X.Neg(\varphi)$, and $Neg(\nu X.\varphi) = \mu X.Neg(\varphi)$

Define the unique *empty environment* as $\perp\colon \emptyset \rightarrow \mathbb{S}$. Then we have:

Theorem 1. *For all closed* NT-μ *formulas* φ, $[\![\varphi]\!]_\perp = \mathbb{S} \setminus [\![Neg(\varphi)]\!]_\perp$.

Proof. For an environment $\mathcal{E}$, let $Neg(\mathcal{E})$ be the environment such that for all variables X, $Neg(\mathcal{E})(X) = \mathbb{S} \setminus \mathcal{E}(X)$. Also, for a summary $\mathbf{s} = \langle s, U_1, \ldots, U_k\rangle$, define $Flip(\mathbf{s})$ to be the summary $\langle s, ME(s) \setminus U_1, \ldots, ME(s) \setminus U_k\rangle$. Thus, a leaf is colored i in $Flip(\mathbf{s})$ iff it is not colored i in $\mathbf{s}$. We lift the map $Flip$ to sets of summaries in the natural way.

Now, by induction on the structure of φ, we prove a stronger assertion: for an NT-μ formula φ and an environment $\mathcal{E}$, we have $[\![\varphi]\!]_\mathcal{E} = \mathbb{S} \setminus Flip([\![Neg(\varphi)]\!]_{Neg(\mathcal{E})})$. Note that the theorem follows when we restrict ourselves to variable and marker-closed formulas.

Cases $\varphi = X$, $\varphi = p$ and $\varphi = \neg p$ are trivial; the cases $\varphi = \mu X.\varphi'$ and $\varphi = nuX.\varphi'$ are easily shown as well. We handle a few other interesting cases.

Suppose $\varphi = \langle ret\rangle R_i$. In this case, $Flip([\![Neg(\varphi)]\!]_{Neg(\mathcal{E})})$ contains the set of summaries $\mathbf{t} = \langle t, U_1, \ldots, U_k\rangle$ such that for all t' satisfying $t \xrightarrow{ret} t'$, we have $t \notin U_i$. It is easy to see that the claim holds.

If $\varphi = \langle call\rangle\varphi'\{\psi_1, \ldots, \psi_k\}$, then $Flip([\![Neg(\varphi)]\!]_{Neg(\mathcal{E})})$ equals the set of summaries $\mathbf{t} = \langle t, U_1, \ldots, U_k\rangle$ such that the following holds: for all t' satisfying $t \xrightarrow{call} t'$, the summary $\mathbf{t}' = \langle t', V_1, V_2, \ldots, V_m\rangle$, where for all $1 \leq i \leq m$, $V_i = ME(t') \cap \{s' : Flip(\langle s', ME(s') \setminus U_1, \ldots, ME(s') \setminus U_k\rangle) \in [\![Neg(\psi_i)]\!]_{Neg(\mathcal{E})}\}$, satisfies $\mathbf{t}' \in [\![Neg(\varphi')]\!]_{Neg(\mathcal{E})}$. Using the induction hypothesis first for the ψ_i-s and then for φ', we can now obtain our claim.

$\square$

Second, note that the semantics of closed NT-μ formulas is independent of the environment; customarily, we will evaluate such formulas in the empty environment $\perp$. More importantly, the semantics of such a formula φ does not depend on current color assignments; in other words, for all $\mathbf{s} = \langle s, U_1, U_2, \ldots, U_k \rangle$, $\mathbf{s} \in [\![\varphi]\!]_\perp$ iff $\langle s \rangle \in [\![\varphi]\!]_\perp$. Consequently, when φ is closed, we can infer that "node s satisfies φ" from "summary $\mathbf{s}$ satisfies φ."

Third, every NT-μ formula $\varphi(X)$ with a free variable X can be viewed as a map $\varphi(X) : 2^{\mathbb{S}} \to 2^{\mathbb{S}}$ defined as follows: for all environments $\mathcal{E}$ and all summary sets $\mathbf{S} \subseteq \mathbb{S}$, $\varphi(X)(\mathbf{S}) = [\![\varphi(X)]\!]_{\mathcal{E}[X:=\mathbf{S}]}$. Then we have:

Proposition 1. *The map $\varphi : 2^{\mathbb{S}} \to 2^{\mathbb{S}}$ is monotonic— i.e., if $\mathbf{S} \subseteq \mathbf{S}' \subseteq \mathbb{S}$, then we have $\varphi(\mathbf{S}) \subseteq \varphi(\mathbf{S}')$.*

It is not hard to verify that this map is monotonic, and that therefore, by the Tarski-Knaster theorem, its least and greatest fixed points exist. The formulas $\mu X.\varphi(X)$ and $\nu X.\varphi(X)$ respectively evaluate to these two sets. From Tarski-Knaster, we also know that for a NT-μ formula φ with one free variable X, the set $[\![\mu X.\varphi]\!]_\perp$ lies in the sequence of summary sets $\emptyset, \varphi(\emptyset), \varphi(\varphi(\emptyset)), \ldots$, and that $[\![\nu X.\varphi]\!]_\perp$ is a member of the sequence $\mathbb{S}, \varphi(\mathbb{S}), \varphi(\varphi(\mathbb{S})), \ldots$.

Fourth, a NT-μ formula φ may also be viewed as a map $\varphi : (U_1, U_2, \ldots, U_k) \mapsto S'$, where S' is the set of all nodes s such that $U_1, U_2, \ldots, U_k \subseteq ME(s)$ and the summary $\langle s, U_1, U_2, \ldots, U_k \rangle$ satisfies φ. Naturally, $S' = \emptyset$ if no such s exists. Now, while a NT-μ formula can demand that the color of a return from the current context is i, it cannot assert that the color of a return *must not be* i (i.e., there is no formula of the form, say, $\langle ret \rangle \neg R_i$). It follows that the output of the above map will stay the same if we grow any of the sets U_i of matching returns provided as input. Formally, we have:

Proposition 2. *Let $\mathbf{s} = \langle s, U_1, \ldots, U_k \rangle$ and $\mathbf{s}' = \langle s, U_1', \ldots U_k' \rangle$ be two summaries such that $U_i \subseteq U_i'$ for all i. Then for every environment $\mathcal{E}$ and every* NT-μ *formula φ, $\mathbf{s}' \in [\![\varphi]\!]_{\mathcal{E}}$ if $\mathbf{s} \in [\![\varphi]\!]_{\mathcal{E}}$.*

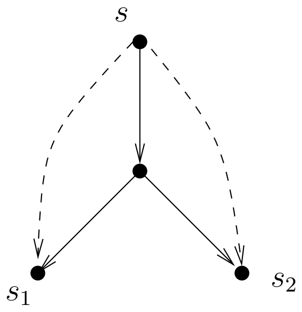

Figure 3.3: Negated return conditions

Such monotonicity over markings has an interesting ramification. Let us suppose that in the semantics clauses for formulas of the form $\langle call \rangle \varphi' \{\psi_1, \psi_2, \ldots, \psi_k\}$ and $[call]\varphi'\{\psi_1, \psi_2, \ldots, \psi_k\}$, we allow $\mathbf{t} = \langle t, V_1, \ldots, V_k \rangle$ to be *any k-colored summary* such that (1) $\mathbf{t} \in [\![\varphi']\!]_{\mathcal{E}}$, and (2) for all i and all $s' \in V_i$, $\langle s', U_1 \cap ME(s'), U_2 \cap ME(s'), \ldots, U_k \cap ME(s') \rangle \in [\![\psi_i]\!]_{\mathcal{E}}$. Intuitively, from such a summary, one can grow the sets U_i to get the "maximal" $\mathbf{t}$ that we used in these two clauses. From the above discussion, NT-μ and this modified logic have equivalent semantics.

Finally, let us see what would happen if we did allow formulas of form $\langle ret \rangle \neg R_i$, which holds at a summary $\langle s, U_1, \ldots, U_k \rangle$ if and only if there is an edge $s \xrightarrow{ret} t$ such that $t \notin U_i$. In other words, such a formula permits us to state what *must not* hold at a colored matching exit in addition to what must. It turns out that formulas involving the above need not be monotonic, and hence their fixpoints may not exist. To see why, consider the formula $\varphi = \langle call \rangle(\langle ret \rangle R_1 \wedge \langle ret \rangle(\neg R_1))\{X\})$ and the nested tree in Figure 3.3. Let $\mathbf{S_1} = \{\langle s_1 \rangle\}$, and $\mathbf{S_2} = \{\langle s_1 \rangle, \langle s_2 \rangle\}$. Viewing φ as a map $\varphi : 2^{\mathbb{S}} \to 2^{\mathbb{S}}$, we see that: (1) $\varphi(\mathbf{S_2}) = \emptyset$, and (2) $\varphi(\mathbf{S_1}) = \langle s \rangle$.

Thus, even though $\mathbf{S_1} \subseteq \mathbf{S_2}$, we have $\varphi(\mathbf{S_1}) \neg \subseteq \varphi(\mathbf{S_2})$. In other words, the monotonicity property breaks down.

3.2.3 Bisimulation closure

Bisimulation is a fundamental relation in the analysis of labeled transition systems. The equivalence induced by a variety of branching-time logics, including the μ-calculus, coincides with bisimulation. In this section, we study the equivalence induced by NT-μ, that is, we want to understand when two nodes satisfy the same set of NT-μ formulas.

Consider two nested trees $\mathcal{T}_1$ and $\mathcal{T}_2$ with node sets S_1 and S_2 (we can assume that the sets S_1 and S_2 are disjoint) and node labeling maps λ_1 and λ_2. Let $S = S_1 \cup S_2$ (we can assume that the sets S_1 and S_2 are disjoint), and let λ denote the labeling of S as given by λ_1 and λ_2. Also, we denote by $\mathbb{S}$ the set of all summaries in $\mathcal{T}_1$ and $\mathcal{T}_2$.

The *bisimulation relation* $\sim \ \subseteq S \times S$ is the greatest relation such that whenever $s \sim t$ holds, we have:

1. $\lambda(s) = \lambda(t)$,

2. for $a \in \{call, ret, loc\}$ and for every edge $s \xrightarrow{a} s'$, there is an edge $t \xrightarrow{a} t'$ such that $s' \sim t'$, and

3. for $a \in \{call, ret, loc\}$ and for every edge $t \xrightarrow{a} t'$, there is an edge $s \xrightarrow{a} s'$ such that $s' \sim t'$.

Let r_1 and r_2 be the roots of $\mathcal{T}_1$ and $\mathcal{T}_2$ respectively. We write $\mathcal{T}_1 \sim \mathcal{T}_2$ if $r_1 \sim r_2$.

NT-μ is interpreted over summaries, so we need to lift the bisimulation relation to summaries. A summary $\langle s, U_1, \ldots U_k \rangle \in \mathbb{S}$ is said to be *bisimulation-closed* if for every pair $u, v \in ME(s)$ of matching exits of s, if $u \sim v$, then for each $1 \leq i \leq k$, $u \in U_i$ precisely when $v \in U_i$. Thus, in a bisimulation-closed summary, the marking does not distinguish among bisimilar nodes, and thus, return formulas (formulas of the form $\langle ret \rangle R_i$ and $[ret] R_i$) do not disntinguish among bisimilar nodes. Two bisimulation-closed summaries $\mathbf{s} = \langle s, U_1, \ldots, U_k \rangle$ and $\mathbf{t} = \langle t, V_1, \ldots, V_k \rangle$ in $\mathbb{S}$ and having the same number of colors are said to be *bisimilar*, written $\mathbf{s} \sim \mathbf{t}$, iff $s \sim t$, and for each $1 \leq i \leq k$, for all $u \in ME(s)$ and $v \in ME(t)$, if $u \sim v$, then $u \in U_i$ precisely when $v \in V_i$. Thus, roots of bisimilar summaries are bisimilar and the corresponding markings are unions of the same equivalence classes of the partitioning of the matching exits induced by bisimilarity. Note that every 0-ary summary is bisimulation-closed, and bisimilarity of 0-ary summaries coincides with bisimilarity of their roots.

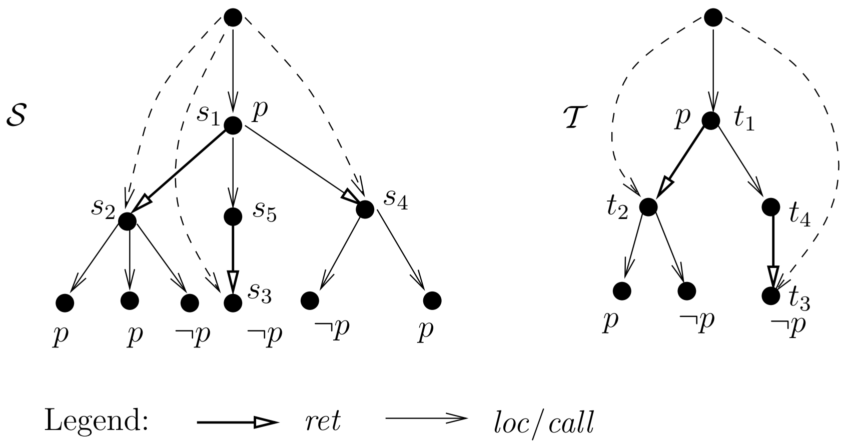

Figure 3.4: Bisimilarity.

Consider the nested trees $\mathcal{S}$ and $\mathcal{T}$ in Figure 3.4. We have named the nodes s_1, s_2, t_1, t_2 etc. and labeled some of them with proposition p. Note that $s_2 \sim s_4$, hence the summary $\langle s_1, \{s_2\}, \{s_4\}\rangle$ in $\mathcal{S}$ is not bisimulation-closed. Now consider the bisimulation-closed summaries $\langle s_1, \{s_2, s_4\}, \{s_3\}\rangle$ and $\langle t_1, \{t_2\}, \{t_3\}\rangle$. By our definition they are bisimilar. However, the (bisimulation-closed) summaries $\langle s_1, \{s_2, s_4\}, \{s_3\}\rangle$ and $\langle t_1, \{t_3\}, \{t_2\}\rangle$ are not.

We now want to prove that bisimilar summaries satisfy the same NT-μ formulas. For an inductive proof, we need to consider the environment also. We assume that the environment $\mathcal{E}$ maps NT-μ variables to subsets of $\mathbb{S}$ (the union of the sets of summaries of the disjoint structures). Such an environment is said to be *bisimulation-closed* if for every variable X, and for every pair of bisimilar summaries $\mathbf{s} \sim \mathbf{t}$, $\mathbf{s} \in \mathcal{E}(X)$ precisely when $\mathbf{t} \in \mathcal{E}(X)$.

Lemma 1. *If $\mathcal{E}$ is a bisimulation-closed environment and φ is a* NT-μ *formula, $[\![\varphi]\!]_{\mathcal{E}}$ is bisimulation-closed.*

Proof. The proof is by induction on the structure of the formula φ. Consider two bisimulation-closed bisimilar summaries $\mathbf{s} = \langle s, U_1, \ldots U_k\rangle$ and $\mathbf{t} = \langle t, V_1, \ldots V_k\rangle$, and a bisimulation-closed environment $\mathcal{E}$. We want to show that $\mathbf{s} \in [\![\varphi]\!]_{\mathcal{E}}$ precisely when $\mathbf{t} \in [\![\varphi]\!]_{\mathcal{E}}$.

If φ is a proposition or negated proposition, the claim follows from bisimilarity of nodes s and t. When φ is a variable, the claim follows from bisimulation closure of $\mathcal{E}$. We consider a few interesting cases.

Suppose $\varphi = \langle \mathit{ret}\rangle R_i$. $\mathbf{s}$ satisfies φ precisely when s has a return-edge to some node s' in U_i. Since s and t are bisimilar, this can happen precisely when t has a return edge to a node t' bisimilar to s', and from definition of bisimilar summaries, t' must be in V_i, and thus $\mathbf{t}$ must satisfy φ.

Suppose $\varphi = \langle \mathit{call}\rangle\varphi'\{\psi_1, \ldots \psi_m\}$. Suppose $\mathbf{s}$ satisfies φ. Then there is a call-successor s' of s such that $\langle s', U'_1, \ldots U'_m\rangle$ satisfies φ', where $U'_i = \{u \in \mathit{ME}(s') \mid \langle u, U_1 \cap \mathit{ME}(u), \ldots U_k \cap \mathit{ME}(u)\rangle \in [\![\psi_i]\!]_{\mathcal{E}}\}$. Since s and t are bisimilar, there exists a call-successor t' of t such that $s' \sim t'$. For each $1 \leq i \leq m$, let $V'_i = \{v \in \mathit{ME}(t') \mid \exists u \in U'_i.\, u \sim v\}$. Verify that the summaries $\langle s', U'_1, \ldots U'_m\rangle$ and $\langle t', V'_1, \ldots V'_m\rangle$ are bisimilar. By induction hypothesis, $\langle t', V'_1, \ldots V'_m\rangle$ satisfies φ'. Also, for each $v \in V'_i$, for $1 \leq i \leq m$, the summary $\langle v, V_1 \cap \mathit{ME}(v), \ldots V_k \cap \mathit{ME}(v)\rangle$ is bisimilar to $\langle u, U_1 \cap \mathit{ME}(u), \ldots U_k \cap \mathit{ME}(u)\rangle$, for some $u \in U_i$, and hence, by induction hypothesis, satisfies ψ_i. This establishes that $\mathbf{t}$ satisfies φ.

To handle the case $\varphi = \mu X.\varphi'$, let $X_0 = \emptyset$. For $i \geq 0$, let $X_{i+1} = [\![\varphi']\!]_{\mathcal{E}[X:=X_i]}$. Then $[\![\varphi]\!]_{\mathcal{E}} = \cup_{i\geq 0} X_i$. Since $\mathcal{E}$ is bisimulation closed, and X_0 is bisimulation-closed, by induction, for $i \geq 0$, each X_i is bisimulation-closed, and so is $[\![\varphi]\!]_{\mathcal{E}}$. $\square$

As a corollary, we get:

Corollary 1. *If $\mathcal{T}_1 \sim \mathcal{T}_2$, then for every closed* NT-μ *formula φ, $\mathcal{T}_1 \models \varphi$ precisely when $\mathcal{T}_2 \models \varphi$.*

The proof also shows that to decide whether a nested tree satisfies a closed NT-μ formula, during the fixpoint evaluation, one can restrict attention only to bisimulation-closed summaries. In other words, we can redefine the semantics of NT-μ so that the set $\mathbb{S}$ of summaries contains only bisimulation-closed summaries. It also suggests that to evaluate a closed NT-μ formula over a nested tree, one can reduce the nested tree by collapsing bisimilar nodes as in the case of classical model checking.

If the two nested trees $\mathcal{T}_1$ and $\mathcal{T}_2$ are not bisimilar, then there exists a μ-calculus formula (in fact, of the much simpler Hennessy-Milner modal logic, which does not involve any fixpoints) that is satisfied at the roots of only one of the two trees. This does not immediately yield a NT-μ formula that distinguishes the two trees because NT-μ formulas cannot assert requirements across return-edges in a direct way. However, as we show in Chapter 4 via an automata-theoretic proof, every closed formula of the μ-calculus may be converted into an equivalent formula in NT-μ. Thus, two nested trees satisfy the same set of closed NT-μ formulas precisely when they are bisimilar.

Let us now consider two arbitrary nodes s and t (in the same nested tree, or in two different nested trees). When do these two nodes satisfy the same set of closed NT-μ formulas? From the arguments so far, bisimilarity is sufficient. However, the satisfaction of a closed NT-μ formula at a node s in a nested tree $\mathcal{T}$ depends solely on the subtree rooted at s that is truncated at the matching exits of s. In fact, the full subtree rooted at s may not be fully contained in a nested tree, as it can contain excess returns. As a result, we define the notion of a *nested subtree* rooted at s as the subgraph obtained by taking the tree rooted at s and deleting the nodes in $ME(s)$ along with the subtrees rooted at them and the return-edges leading to them (the jump-edge relation is restricted in the natural way).

For instance, in Figure 3.4, $\mathcal{S}_{s_1}$ comprises nodes s_1 and s_5 and the *loc*-edge connecting them. It is easy to check that for a node s in a nested tree $\mathcal{T}$ and a closed NT-μ formula φ, the summary $\langle s \rangle$ satisfies φ in the original nested tree precisely when $\mathcal{T}_s$ satsifies φ. If s and t are not bisimilar, and the non-bisimilarity can be established within the nested subtrees $\mathcal{T}_s$ and $\mathcal{T}_t$ rooted at these nodes, then some closed NT-μ formula can distinguish them.

Theorem 2. *Two nodes s and t satisfy the same set of closed* NT-μ *formulas precisely when $\mathcal{T}_s \sim \mathcal{T}_t$.*

3.3 Specifying requirements

In this section, we explore how to use NT-μ as a specification language. On one hand, we will see how NT-μ and classical temporal logics differ fundamentally in style of expression; on the other, we will express properties not expressible in logics like the μ-calculus. The example program from Chapter 2 (reproduced, along with the corresponding nested tree, in Figure 3.5) will be used to illustrate some of our specifications. Also, because fixpoint formulas are typically hard to read, we will define some syntactic sugar for NT-μ using CTL-like temporal operators.

Reachability

Let us express in NT-μ the reachability property *Reach* that says: "a node t satisfying proposition p can be reached from the current node s before the current context ends." As a program starts with an empty stack frame, we may omit the restriction about the current context if s models the initial program state.
Now consider a nontrivial witness π for *Reach* that starts with an edge $s \xrightarrow{call} s'$. There are two possibilities: (1) a node satisfying p is reached in the new context or a context called transitively from it, and (2) a matching exit s'' of s' is reached, and at s'', *Reach* is once again satisfied.

To deal with case (2), we mark a matching exit that leads to p by color 1. Let X store the set of summaries of form $\langle s'' \rangle$, where s'' satisfies *Reach*. Then we want the summary $\langle s, ME(s) \rangle$ to satisfy $\langle call \rangle \varphi' \{X\}$, where φ' states that s' can reach one of its matching exits of color 1. In case (1), there is no return requirement (we do not need the original call to return), and we simply assert $\langle call \rangle X\{\}$.

Before we get to φ', note that the formula $\langle loc \rangle X$ captures the case when π starts with a local transition. Combining the two cases and using CTL-style notation (we write $EF_c\ p$ to denote "p is true before the end of the current context ends"), the formula we want is

$$EF_c\ p = \mu X.(p \vee \langle loc \rangle X \vee \langle call \rangle X\{\} \vee \langle call \rangle \varphi' \{X\}).$$

Now observe that φ' also expresses reachability, except: (1) its target needs to satisfy $\langle ret \rangle R_1$, and (2) this target needs to lie in the *same procedural context* as s'. In other words, we want to express what we call *local reachability* of $\langle ret \rangle R_1$. It is easy to verify that

$$\varphi' = \mu Y.(\langle ret \rangle R_1 \vee \langle loc \rangle Y \vee \langle call \rangle Y\{Y\}).$$

We cannot merely substitute p for $\langle ret \rangle R_1$ in φ' to express local reachability of p. However, a formula $EF_c^l\ p$ for this property is easily obtained by restricting the formula $EF_c\ p$:

$$EF_c^l\ p = \mu X.(p \vee \langle loc \rangle X \vee \langle call \rangle \varphi' \{X\}).$$

Generalizing, we can allow p to be any NT-μ formula that keeps $EF_c\ p$ and $EF_c^l\ p$ closed.

For example, consider the nested tree in Figure 3.5 that models the unfolding of the program in the same figure. The propositional labeling is as in Chapter 2. In that case, $EF_c^l\ rd$ and $EF_c\ wr$ are true at the control point right before the recursive call in L2 in the top-level invocation of *foo* (node s in the figure); however, $EF_c^l wr$ is not.

It is now easy to verify that the formula $AF_c\ p$, which states that "along *all* paths from the current node, a node satisfying p is reached before the current context terminates," is given by

$$AF_c\, p = \mu X.(p \vee ([loc]X \wedge [call]\varphi''\{X\})),$$

where φ'' demands that a matching exit colored 1 be reached along all local paths:

$$\varphi'' = \mu Y.(p \vee ([ret]R_1 \wedge [loc]Y \wedge [call]Y\{Y\})).$$

As in the previous case, we can define a corresponding operator AF^l_c that asserts local reachability along all paths. For instance, in Figure 3.5, $AF^l_c\, rd$ does *not* hold at node s (as the program can skip its while-loop altogether).

Note that the highlight of this approach to specification is the way we split a program unfolding along procedure boundaries, specify these "pieces" modularly, and plug the summary specifications so obtained into their call sites. This "interprocedural" reasoning distinguishes it from logics such as the μ-calculus that would reason only about *global* runs of the program.

Also, there is a significant difference in the way fixpoints are computed in NT-μ and the μ-calculus. Consider the fixpoint computation for the μ-calculus formula $\mu X.(p \vee \langle\rangle X)$ that expresses reachability of a node satisfying p. The semantics of this formula is given by a set S_X of nodes which is computed iteratively. At the end of the i-th step, S_X comprises nodes that have a path with at most $(i-1)$ transitions to a node satisfying p. Contrast this with the evaluation of the outer fixpoint in the NT-μ formula $EF_c\ p$. Assume that φ' (intuitively, the set of "jumps" from calls to returns") has already been evaluated, and consider the set $\mathbf{S_X}$ of summaries for $EF_c\ p$. At the end of the i-th phase, this set contains all $\mathbf{s} = \langle s \rangle$ such that s has a path consisting of $(i-1)$ *call* and *loc*-transitions to a node satisfying p. However, because of the subformula $\langle call \rangle \varphi'\{X\}$, it also includes all $\mathbf{s}$ where s reaches p via a path of at most $(i-1)$ local and "jump" transitions. Note how return edges are considered only as part of summaries plugged into the computation.

```
input x;

procedure foo()
{
L1:  write(e);
     if(x)
L2:    foo();
     else
L3:    think;
     while (x)
L4:    read(e);
L5:  return;
}
```

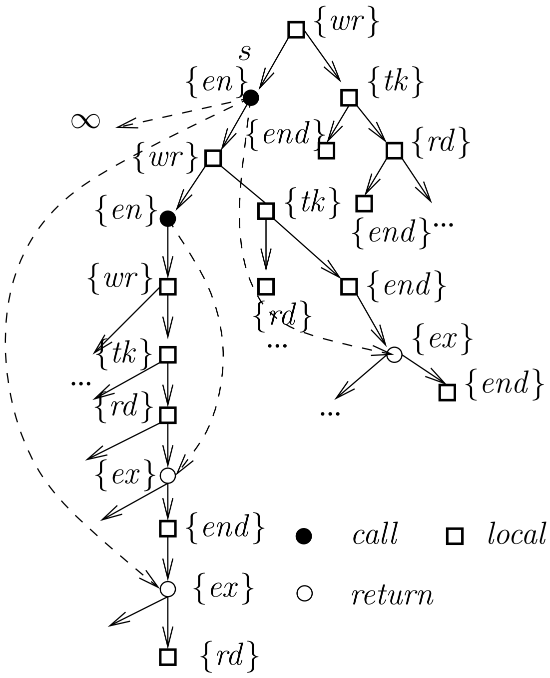

Figure 3.5: An example

Invariance and until

Now consider the *invariance* property "on some path from the current node, property p holds everywhere till the end of the current context." A NT-μ formula $EG_c\ p$ for this is obtained from the identity $EG_c\ p = Neg(AF_c\ Neg(p))$. The formula $AG_c\ p$, which asserts that p holds on each point on each run from the current node, can be written similarly.

Other classic branching-time temporal properties like the *existential weak until* (written as $E(p_1\ W_c\ p_2)$) and the *existential until* ($E(p_1\ U_c\ p_2)$) are also expressible. The former holds if there is a path π from the current node such that p_1 holds at every point on π till it reaches the end of the current context or a node satisfying p_2 (if π doesn't reach either, p_1 must hold all along on it). The latter, in addition, requires p_2 to hold at some point on π. The for-all-paths analogs of these properties ($A(p_1\ U_c\ p_2)$ and $A(p_1\ W_c\ p_2)$) aren't hard to write either.

Neither is it difficult to express local or same-context versions of these properties. Consider the maximal subsequence π' of a program path π from s such that each node of π' belongs to the same procedural context as s. A NT-μ formula $EG^l p$ for *existential local invariance* demands that p holds on some such π', while $AG^l_c p$ asserts the same for all π'. Similarly, we can define existential and universal *local until* properties, and corresponding NT-μ formulas $E(p_1\ U^l_c\ p_2)$ and $A(p_1\ U^l_c\ p_2)$. For instance, in Figure 3.5, $E(\neg wr\ U^l_c\ rd)$ holds at node s (whereas $E(\neg wr\ U_c\ rd)$ does not). "Weak" versions of these formulas are also written with ease. For instance, it is easy to verify that we can write generic existential, local, weak until properties as

$$E(p_1\ W^l_c\ p_2) = \nu X.((p_1 \vee p_2) \wedge (p_2 \vee \langle loc \rangle X \vee \langle call \rangle \varphi'\{X\})),$$

where φ' asserts local reachability of $\langle ret \rangle R_1$ as before.

Interprocedural dataflow analysis

It is well-known that many classic dataflow analysis problems can be reduced to temporal logic model-checking over program abstractions [Ste91, Sch98]. For example, consider the problem of finding *very busy expressions* in a program that arises in compiler optimization. An expression e is said to be very busy at a program point s if every path from s must evaluate e before any variable in e is redefined. Let us first assume that all variables are in scope all the time along every path from s. Now label every node in the program's unfolding immediately preceding a statement evaluating e by a proposition $use(e)$, and every node representing a program state about to redefine a variable in e by $mod(e)$. For example, if e is as in the program in Figure 3.5, every node labeled wr in the corresponding nested tree is also labeled $mod(e)$, and every node labeled rd is also labeled $use(e)$.

Because of loops in the flow graph, we would not expect every path from s to eventually satisfy $use(e)$; however, we can demand that each point in such a loop will have a path to a loop exit from where a use of e would be reachable. Then a

NT-μ formula that demands that e is very busy at s is

$$A((EF_c\ use(e) \wedge \neg mod(e))\ W_c\ use(e)).$$

Note that this property uses the power of NT-μ to reason about branching time. However, complications arise if we are considering interprocedural paths and e has local as well as global variables. Note that if e in Figure 3.5 contains global variables, then it is not very busy at the point right before the recursive call to *foo*. This is because e may be written in the new context. However, if e only contains local variables, then this modification, which happens in an invoked procedural context, does not affect the value of e in the original context. While facts involving global variables and expressions flow through program paths across contexts, data flow involving local variables follow program paths within the same context.
Local temporal properties are useful in capturing these two different types of data flow. Let us handle the general case, where the expression e may have global as well as local variables. define two propositions $mod_g(e)$ and $mod_l(e)$ that are true at points where, respectively, a global or a local variable in e is modified. The NT-μ property we assert at s is

$$\nu X.(((EF_c^l\ use(e)) \wedge \neg mod_g(e) \wedge \neg mod_l(e)) \vee use(e)) \wedge (use(e) \vee ([loc]X \wedge [call]\psi\{X, tt\})),$$

where the formula ψ tracks global variables in new contexts:

$$\psi = \mu Y.(\neg mod_g(e) \wedge (([ret]R_1 \wedge \langle ret\rangle R_2) \vee ([call]Y\{Y, tt\} \wedge [loc]Y))).$$

Note the use of the formula $\langle ret\rangle R_2$ to ensure that $[ret]R_1$ is not vacuously true.

Pushdown specifications

The domain where NT-μ stands out most clearly from previously studied fixpoint calculi is that of *pushdown specifications*, i.e., specifications involving the program stack. We have already introduced a class of such specifications expressible in NT-μ: that of local temporal properties. For instance, the formula $EF^l_c p$ needs to track the program stack to know whether a reachable node satisfying p is indeed in the initial calling context. Some such specifications have previously been discussed in context of the temporal logic CARET [AEM04]. On the other hand, it is well-known that the modal μ-calculus is a *regular* specification language (i.e., it is equivalent in expressiveness to a class of finite-state tree automata), and cannot reason about the stack in this way. We have already seen an application of these richer specifications in program analysis. In the rest of this section, we will see more of them.

Nested formulas and stack inspection

Interestingly, we can express certain properties of the stack just by nesting NT-μ formulas for (non-local) reachability and invariance. To understand why, recall that NT-μ formulas for reachability and invariance only reason about nodes appearing before the end of the context where they were asserted. Now let us try to express a *stack inspection property* such as "if procedure `foo` is called, procedure `bar` must not be on the call stack." Specifications like this have previously been used in research on software security [JMT99, EKS03], and are enforced at runtime in the Java or .NET stack inspection framework. However, because a program's stack can be unbounded, they are not expressible by regular specifications like the μ-calculus. While the temporal logic CARET can express such properties, it requires a past-time operator called *caller* to do so. To express this property in NT-μ, we define propositions $c_{\texttt{foo}}$ and $c_{\texttt{bar}}$ that respectively hold at every call site for `foo` and `bar`. Now, assuming control starts in `foo`, consider the formula

$$\varphi = EF_c(c_{\texttt{bar}} \wedge \langle call \rangle (EF_c\ c_{\texttt{foo}})\{\}).$$

This formula demands a program path where, first, `bar` is called (there is no return requirement), and then, before that context is popped off the stack, a call site for `foo` is reached. It follows that the property we are seeking is $Neg(\varphi)$.

Other stack inspection properties expressible in NT-μ include "when procedure `foo` is called, all procedures on the stack must have the necessary privilege." Like the previous requirement, this requirement protects a a privileged callee from a malicious caller. However, NT-μ also comes in handy to express properties that protect the caller from the callee. For one such scenario [AF03], consider a malicious method A which, via side-effects or exceptional return conditions, may compromise the security of methods that the caller method B calls subsequently. To prevent such a scenario, we may assert requirements such as "If A has ever been on the stack, do not execute the sensitive operation X." Note that stack inspection cannot handle this specification. This is an example of a *dynamic security constraint* (mentioned but not formalized in [JMT99]), which combine reasoning about the program stack with reasoning about the global evolution of the program, allowing privileges of procedures to change dynamically depending on the privileges used so far.

Stack overflow

Stack overflow, caused by unbounded recursion, is a serious security vulnerability in programs written in C-like languages. NT-μ can specify requirements that safeguard against such errors. Once again, nested modalities come handy. Suppose we assert $AG_c(\langle call \rangle \mathit{ff}\{\})$ throughout every context reached through k calls in succession without intervening returns (this can be kept track of using a k-length chain of $\langle call \rangle$ modalities). This will disallow further calls, bounding the stack to height k.

Other specifications for stack boundedness include: "every call in every program execution eventually returns." This property requires the program stack to be empty infinitely often. Though this requirement does not say how large the stack may get—even if a call returns, it may still overflow the stack at some point. Further, in certain cases, a call may not return because of cycles introduced by abstraction. However, it does rule out infinite recursive loops in many cases; for instance, the program in Fig. 3.5 will fail this property because of a real recursive cycle. We capture it by asserting AG_c *Termin* at the initial program point, where

$$Termin = [call](AF^l_c(\langle ret\rangle R_1))\{tt\}.$$

Preconditions and postconditions

For a program state s, let us consider the set $Jmp(s)$ of nodes to which a call from s may return. Then the requirement: "property p holds at some node in $Jmp(s)$" is captured by the NT-μ formula $\langle jump\rangle p = \langle call\rangle(EF^l_c\ \langle ret\rangle R_1)\{p\}$. The dual formula $[jump]p$, which requires p to hold at all such jump targets, is also easily constructed. An immediate application of this is to encode the partial and total correctness requirements popular in formalisms like Hoare logic and JML [BCC$^+$03]. A partial correctness requirement for a procedure `A` asserts that if precondition *Pre* is satisfied when `A` is called, then if `A` terminates, postcondition *Post* holds upon return. Total correctness, additionally, requires `A` to terminate. These requirements cannot be expressed using regular specifications. In NT-μ, let us say that at every call site to procedure `A`, proposition c_A holds. Then a formula for partial correctness, asserted at the initial program state, is

$$AG_c((Pre \wedge c_A) \Rightarrow [jump]Post).$$

Total correctness is expressed as

$$AG_c((Pre \wedge c_A) \Rightarrow (Termin \wedge [jump]Post)).$$

Access control

The ability of NT-μ to handle local and global variables simultaneously is useful in other domains, e.g., access control. Consider a procedure `A` that can be called with a high or low privilege, and suppose we have a rule that `A` can access a database (proposition *access* is true when it does) only if it is called with a high privilege (*priv* holds when it is). It is tempting to write a property $\varphi = \neg priv \Rightarrow AG_c\ (\neg access)$

to express this requirement. However, a context where `A` has low privilege may lead to another where `A` has high privilege via a recursive invocation, and φ will not let `A` access the database even in this new context. The formula we are looking for is really $\varphi' = \neg priv \Rightarrow AG_c^l\,(\neg access)$, asserted at every call site for `A`.

Multiple return conditions

As we shall see in Section 3.5, the theoretical expressiveness of NT-μ depends on the fact that we can pass multiple return conditions as "parameters" to NT-μ call formulas. We can also use these parameters to remember events that happen within the scope of a call and take actions accordingly on return.

To see how, we go back to Figure 3.5, and observe that in any particular invocation of *foo*, it is possible to exit the routine (1) having read the value of e that was written in this invocation, and (2) not having read this value. Suppose that we demand that in case (2), the expression e must be read at least once before the end of the current context— that is, the value written in the last write must be read by that point. We do not require this in case (1)— in this case control may skip the loop. In addition, let us require that every path in an invocation of *foo* returns and that e is written at least once in this path.

We express these requirements by asserting the NT-μ formula φ at the program point right before the recursive call to *foo*:

$$\varphi = \langle call \rangle \psi' \{tt, EF_c\ rd\},$$

where ψ' is a fixed-point property that states that: each path in the new context must either:

- see a node labeled wr followed, not necessarily immediately, by a node labeled rd, and then read $\langle ret \rangle R_1$ without seeing wr again,
- see a node labeled wr and then not see a node labeled rd till a node satisfying $\langle ret \rangle R_2$ is reached.

3.4 Model checking

In this section, we introduce the problem of model checking NT-μ over unfoldings of nested state machines (NSMs). Our primary result is an iterative, symbolic decision procedure to solve this problem. Appealingly, this algorithm follows directly from the operational semantics of NT-μ and has the same complexity as the best algorithms for model checking CTL or the alternation-free μ-calculus over similar abstractions. We also show a matching lower bound.

For a specification given by a (closed) NT-μ formula φ and an NSM $\mathcal{M}$ abstracting a program (recall Section 2.2), the *model checking problem* is to determine if $\mathcal{T}(\mathcal{M})$ satisfies φ. We will now offer an algorithm for this problem.

Let V be the set of vertices of $\mathcal{M}$, and consider a node s in the execution tree $\mathcal{T}^V(\mathcal{M})$ of $\mathcal{M}$ (defined in Chapter 2). The set $ME(s)$, as well as the return-formulas that hold at a summary $\mathbf{s}$ rooted at s, depend on states at call nodes on the path from the root to s. However, we observe that the history of call-nodes up to s is relevant to a formula only because they may be consulted by return-nodes in the future, and no formula interpreted at $\mathbf{s}$ can probe "beyond" the nodes in $ME(s)$. Thus, so far as satisfaction of a formula goes, we are only interested in the *last* "pending" call-node; in fact, the state of the automaton at this node is all that we need to record about the past.

Let us now try to formalize this intuition. First we define the *unmatched call-ancestor* $Anc(s)$ of a node s in a nested tree $\mathcal{T}$. Consider the tagged tree of $\mathcal{T}$, and recall the definition of a balanced word over tags (given in Section 2.1). If $t = Anc(s)$, then we require that $t \xrightarrow{call} t'$ for some node t' such that in the tagged tree of $\mathcal{T}$, there is a path π from t' to s such that the sequence of edge labels along π forms a balanced word. Note that every node in a nested tree has at most one unmatched call-ancestor. If a node s does not have such an ancestor, we set $Anc(s) = \perp$.

Now let us consider two k-colored summaries $\mathbf{s} = \langle s, U_1, U_2, \ldots, U_k \rangle$ and $\mathbf{s}' = \langle s', U'_1, U'_2, \ldots, U'_k \rangle$ in the execution tree $\mathcal{T}^V(\mathcal{M}) = (T, \hookrightarrow, \lambda)$ of the NSM $\mathcal{M}$, and let $Anc(s) = t$ and $Anc(s') = t'$, where t, t' can be nodes or the symbol $\perp$ (note that if we have $Anc(s) = \perp$, then $ME(s) = \emptyset$, so that $U_i = \emptyset$ for all i).

Now we say $\mathbf{s}$ and $\mathbf{s}'$ are $\mathcal{M}$-*equivalent* (written as $\mathbf{s} \equiv_{\mathcal{M}} \mathbf{s}'$ or simply $\mathbf{s} \equiv \mathbf{s}'$ when $\mathcal{M}$ is clear from the context) if:

- $\lambda(s) = \lambda(s')$;
- either $t = t' = \perp$, or $\lambda(t) = \lambda(t')$;
- for each $1 \leq i \leq k$, there is a bijection $\Omega_i : U_i \to U_i'$ such that for all $u \in U_i$, we have $\lambda(u) = \lambda(\Omega_i(u))$.

It is easily seen that the relation $\equiv$ is an equivalence. Let us call a set $\mathbf{S}$ of summaries in $\mathcal{T}^V(\mathcal{M})$ $\mathcal{M}$-*equivalence-closed* (written $\equiv$-closed) if for any two summaries $\mathbf{s}$ and $\mathbf{s}'$ such that $\mathbf{s} \equiv \mathbf{s}'$, we have $\mathbf{s} \in \mathbf{S}$ iff $\mathbf{s}' \in \mathbf{S}$. In other words, such a set is the union of a certain number of equivalence classes induced by $\equiv$. Let us call an environment $\mathcal{E}$ $\equiv$-*closed* if for every variable X, the set $\mathcal{E}(X)$ is $\equiv$-closed.

Note that the above definitions involve summaries in the *execution tree* $\mathcal{T}^V(\mathcal{M})$. However, each summary in the execution tree may also be viewed as a summary in the unfolding $\mathcal{T}(\mathcal{M})$ of $\mathcal{M}$, in which case we can interpret NT-μ formulas involving atomic propositions on them.

Now we prove that:

Lemma 2. *For any $\equiv$-closed environment $\mathcal{E}$,* NT-μ *formula φ, and $\equiv$-equivalent summaries $\mathbf{s}$ and $\mathbf{s}'$, we have $\mathbf{s} \in [\![\varphi]\!]_{\mathcal{E}}^{\mathcal{T}(\mathcal{M})}$ iff $\mathbf{s}' \in [\![\varphi]\!]_{\mathcal{E}}^{\mathcal{T}(\mathcal{M})}$.*

Proof. The proof is by structural induction on the formula φ. Let $\mathbf{s}$ and $\mathbf{s}'$ be $\mathcal{M}$-equivalent summaries as in the definition of $\mathcal{M}$-equivalence above, and let us continue to denote the bijection between the i-colored exits of $\mathbf{s}$ and $\mathbf{s}'$ by $\Omega_i : U_i \to U_i'$. Let us denote by λ the labeling of nodes in the execution tree $\mathcal{T}^V(\mathcal{M})$. We denote $\lambda(s) = \lambda(s')$ by v_0 and $\lambda(t) = \lambda(t')$ by v_1. It suffices to show that assuming $\mathcal{E}$ is $\equiv$-closed, $\mathbf{s} \in [\![\varphi]\!]_{\mathcal{E}} \Rightarrow \mathbf{s}' \in [\![\varphi]\!]_{\mathcal{E}}$.

We handle some interesting cases:

- If $\varphi = X$, then the claim holds as $\mathcal{E}$ is $\mathcal{M}$-equivalence-closed.

- Suppose $\varphi = \mu X.\varphi'$. Let $X_0 = \emptyset$, and for $i \geq 0$, let $X_{i+1} = [\![\varphi']\!]_{\mathcal{E}[X:=X_i]}$. Then $[\![\varphi]\!]_{\mathcal{E}} = \cup_{i\geq 0} X_i$. Since X_0 is $\mathcal{M}$-equivalence-closed, and $\mathcal{E}$ is $\mathcal{M}$-equivalence-closed, and $[\![\varphi']\!]_{\mathcal{E}'}$ is $\mathcal{M}$-equivalence-closed for all $\mathcal{M}$-equivalence-closed environments $\mathcal{E}'$, by induction on i, $[\![\varphi]\!]_{\mathcal{E}}$ is $\mathcal{M}$-equivalence-closed.

- Suppose $\varphi = \langle ret \rangle R_i$. That means that there is some $s \xrightarrow{ret} u$ such that $u \in U_i$. Let $\lambda(u) = v_2$; then $\lambda(\Omega(u)) = v_2$. However, in that case the same transition $(v_0, v_1) \xrightarrow{ret} v_2$ can be "fired" from s' to $\Omega(u)$— i.e., $s' \xrightarrow{ret} \Omega(u)$. In other words $\mathbf{s}' \in [\![\varphi]\!]_{\mathcal{E}}$

- Suppose $\varphi = \langle call \rangle \varphi' \{\psi_1, \ldots, \psi_k\}$. This means that there is some call-edge $s \xrightarrow{call} u$ and a summary $\mathbf{u} = \langle u, V_1, \ldots, V_k \rangle$ satisfying φ' such that for all i and all $w \in V_i$, we have $\mathbf{w} = \langle t, V_1 \cap ME(w), \ldots, V_k \cap ME(w) \rangle$ satisfying ψ_i.

  Now note that, because $\lambda(s) = \lambda(s')$, there is a call-edge $s' \xrightarrow{call} u'$. We can show inductively that there is a bijection $\Omega' : ME(u) \to ME(u')$ such that for any node $w \in ME(u)$, we have $\lambda(w) = \lambda(\Omega'(w))$. The reason is that any path in the tree from s to $ME(s)$ is independent of the labeling $\lambda(w)$ of any node w such that $w \xrightarrow{+} Anc(s)$ (we can prove this using induction). As $\lambda(s) = \lambda(s')$ and $\lambda(Anc(s)) = \lambda(Anc(s'))$, every path in from s to a node in $ME(s)$ is simulated by a unique path from s' to a node in $ME(s')$.

  By restricting Ω' to the sets $V_1, V_2, \ldots \subseteq ME(u)$, we get the bijections $\Omega'_i : V_i \to V'_i$, for $1 \leq i \leq k$. Let us now construct $\mathbf{u}' = \langle u', V'_1, \ldots, V'_k \rangle$. We observe that $\mathbf{u}$ and $\mathbf{u}'$ are $\mathcal{M}$-equivalent. In that case $\mathbf{u}'$ satisfies φ' in environment $\mathcal{E}$. Likewise, for $w \in V_i$ as before, let $w' = \Omega_i(w)$, and construct the summary $\mathbf{w}' = \langle w', V'_1 \cap ME(w'), \ldots, V'_k \cap ME(w') \rangle$. Then $\mathbf{w}$ and $\mathbf{w}'$ are $\mathcal{M}$-equivalent, and both satisfy ψ_i. In that case $\mathbf{s}'$ satisfies φ.

$\square$

Now note that the number of equivalence classes that $\equiv$ induces on the set of summaries is bounded! Each such equivalence class may be represented by a tuple $\langle v, v', V_1, \ldots, V_k \rangle$, where $v \in V$, $v' \in V \cup \{\perp\}$, and $V_i \subseteq V$ for all i—for the class of the summary $\mathbf{s}$ above, for instance, we have $\lambda(s) = v$ and $\lambda(U_i) = V_i$; we also have $\lambda(t) = v'$ in case $t \neq \perp$, and $v' = \perp$ otherwise. Let us call such a tuple a *bounded summary*. The idea behind the model-checking algorithm of NT-μ is that for any formula φ, we can maintain, symbolically, the set of bounded summaries that satisfy it. Once this set is computed, we can compute the set of bounded summaries for formulas defined inductively in terms of φ. This computation follows directly from the semantics of the formula; for instance, the set for the formula $\langle loc \rangle \varphi$ contains all bounded summaries $\langle v, v', V_1, \ldots, V_k \rangle$ such that for some $v'' \in V$, we have $v \xrightarrow{loc} v''$, and, letting V_i'' comprise the elements of V_i that are reachable from v'', $\langle v'', v', V_1'', \ldots, V_k'' \rangle$ satisfies φ.

Let us now define bounded summaries formally. Consider any state u in an NSM $\mathcal{M}$ with state set V. A state u' is said to be the *unmatched call-ancestor state* of state u if there is a node s labeled u in $\mathcal{T}^V(\mathcal{M})$ such that u' is the label of the unmatched call-ancestor of s (we have a predicate $Anc_V(u', u)$ that holds iff this is true). Note that a state may have multiple unmatched call-ancestor states. If there is a node s labeled u in $\mathcal{T}^V(\mathcal{M})$ such that $Anc(s) = \perp$, we set $Anc_V(\perp, u)$.

A state v is a *matching exit state* for a pair (u, u'), where $Anc_V(u', u)$, if there are nodes s, s', t in $\mathcal{T}^V(\mathcal{M})$ such that $t \in ME(s)$, s' is the unmatched call-ancestor of s, and labels of s, s', and t are u, u', and v respectively (a pair $(u, \perp)$ has no matching exit state).

The modeling intuition is that from a program state modeled by NSM state u and a stack with a single frame modeled by the state u', control may reach a u'' in the same context, and then return at the state v via a transition $(u'', u') \xrightarrow{ret} v$. Using well-known techniques for pushdown models [ABE+05], we can compute, given a state u, the set of u' such that $Anc_V(u', u)$, and for every member u' of the latter, the set $MES(u, u')$ of matching exit states for (u, u'), in time polynomial in the size of $\mathcal{M}$.

Now, let n be the arity of the formula φ in whose model-checking problem we are interested. A *bounded summary* is a tuple $\langle u, u', V_1, \ldots, V_k \rangle$, where $0 \leq k \leq n$, $Anc_V(u', u)$ and for all i, we have $V_i \subseteq MES(u, u')$. The set of all bounded summaries in $\mathcal{M}$ is denoted by BS.

Let $\mathcal{E}_{BS} : Free(\varphi) \to 2^{BS}$ be a *bounded environment* mapping free variables in φ to sets of bounded summaries, and let $\perp_B$ denote the empty environment. We define a map $Eval(\varphi, \mathcal{E}_{BS})$ assigning a set of bounded summaries to a NT-μ formula φ:

- If $\varphi = p$, for $p \in AP$, then $Eval(\varphi, \mathcal{E}_{BS})$ consists of all bounded summaries $\langle u, u', V_1, \ldots, V_k \rangle$ such that $p \in \kappa(u)$ and $k \leq n$.

- If $\varphi = \neg p$, for $p \in AP$, then $Eval(\varphi, \mathcal{E}_{BS})$ consists of all bounded summaries $\langle u, u', V_1, V_2, \ldots, V_k \rangle$ such that $p \notin \kappa(u)$ and $k \leq n$.

- If $\varphi = X$, for $X \in Var$, then $Eval(\varphi, \mathcal{E}_{BS}) = \mathcal{E}_{BS}(X)$.

- If $\varphi = \varphi_1 \vee \varphi_2$ then $Eval(\varphi, \mathcal{E}_{BS}) = Eval(\varphi_1, \mathcal{E}_{BS}) \cup Eval(\varphi_2, \mathcal{E}_{BS})$.

- If $\varphi = \varphi_1 \wedge \varphi_2$ then $Eval(\varphi, \mathcal{E}_{BS}) = Eval(\varphi_1, \mathcal{E}_{BS}) \cap Eval(\varphi_2, \mathcal{E}_{BS})$.

- If $\varphi = \langle call \rangle\ \varphi'\{\psi_1, ..., \psi_m\}$, then $Eval(\varphi, \mathcal{E}_{BS})$ consists of all bounded summaries $\langle u, u', V_1, \ldots, V_k \rangle$ such that for some transition $u \xrightarrow{call} u''$ of $\mathcal{M}$, we have a bounded summary $\langle u'', u'', V_1', V_2', ..., V_m' \rangle \in Eval(\varphi', \mathcal{E}_{BS})$, and for all $v \in V_i'$, where $i = 1, \ldots, m$, we have $\langle v, u', V_1'', \ldots, V_k'' \rangle \in Eval(\psi_i, \mathcal{E}_{BS})$, where $V_j'' = V_j \cap MES(v, u')$ for all $j \leq k$.

- If $\varphi = [call]\ \varphi'\{\psi_1, ..., \psi_m\}$, then $Eval(\varphi, \mathcal{E}_{BS})$ consists of all bounded summaries $\langle u, u', V_1, \ldots, V_k \rangle$ such that for all u'' such that there is a transition $u \xrightarrow{call} u''$ in $\mathcal{M}$, we have a bounded summary $\langle u'', u'', V_1', V_2', ..., V_m' \rangle \in Eval(\varphi', \mathcal{E}_{BS})$,

FixPoint $(X, \varphi, \mathcal{E}_{BS})$
$X' = Eval(\varphi, \mathcal{E}_{BS})$
if $X' = \mathcal{E}_{BS}(X)$
  then return X'
  else return FixPoint $(X, \varphi', \mathcal{E}_{BS}[X := X'])$

Figure 3.6: Fixpoint computation for NT-μ.

and for all $v \in V_i'$, where $i = 1, \ldots, m$, we have $\langle v, u', V_1'', \ldots, V_k'' \rangle \in Eval(\psi_i, \mathcal{E}_{BS})$, where $V_j'' = V_j \cap MES(v, u')$ for all $j \leq k$.

- If $\varphi = \langle loc \rangle\, \varphi'$, then $Eval(\varphi, \mathcal{E}_{BS})$ consists of all bounded summaries of the form $\langle u, u', V_1 \ldots, V_k \rangle$ such that for some v such that there is a transition $u \xrightarrow{loc} v$, we have $\langle v, u', V_1 \cap MES(v, u'), \ldots, V_k \cap MES(v, u') \rangle \in Eval(\varphi', \mathcal{E}_{BS})$.

- If $\varphi = [loc]\, \varphi'$, then $Eval(\varphi, \mathcal{E}_{BS})$ consists of all bounded summaries of the form $\langle u, u', V_1 \ldots, V_k \rangle$ such that for all v such that there is a transition $u \xrightarrow{loc} v$, we have $\langle v, u', V_1 \cap MES(v, u'), \ldots, V_k \cap MES(v, u') \rangle \in Eval(\varphi', \mathcal{E}_{BS})$.

- If $\varphi = \langle ret \rangle\, R_i$, then $Eval(\varphi, \mathcal{E}_{BS})$ consists of all bounded summaries of the form $\langle u, u', V_1, \ldots, V_k \rangle$ such that for some $u'' \in V_i$, $\mathcal{M}$ has a transition $(u, u') \xrightarrow{ret} u''$.

- If $\varphi = [ret]\, R_i$, then $Eval(\varphi, \mathcal{E}_{BS})$ consists of all bounded summaries of the form $\langle u, u', V_1, \ldots, V_k \rangle$ such that for all transitions of the form $(u, u') \xrightarrow{ret} u''$, we have $u'' \in V_i$.

- If $\varphi = \mu X.\varphi'$, then $Eval(\varphi, \mathcal{E}_{BS}) = FixPoint\ (X, \varphi', \mathcal{E}_{BS}[X := \emptyset])$.

- If $\varphi = \nu X.\varphi'$, then $Eval(\varphi, \mathcal{E}_{BS}) = FixPoint\ (X, \varphi', \mathcal{E}_{BS}\ [X := BS])$.

Here $FixPoint\ (X, \varphi, \mathcal{E}_{BS})$ is a fixpoint computation function that uses the formula φ as a monotone map between subsets of BS, and iterates over variable X. This computation is as in Figure 3.6.

Now we show that to model check a formula φ on an NSM $\mathcal{M}$, we only have to compute $Eval(\varphi, \perp_B)$. Let us first define, for any bounded summary $\mathbf{b} = \langle v, v', V_1, \ldots, V_k \rangle$, the set $Unbound(\mathbf{b})$ of summaries in $\mathcal{T}^V(\mathcal{M})$ that it captures. Let λ be denote the node-labeling map in $\mathcal{T}^V(\mathcal{M})$. A summary $\mathbf{s} = \langle s, U_1, \ldots, U_k \rangle$ is in $Unbound(\mathbf{b})$ iff $\lambda(s) = v$, $\lambda(Anc(s)) = v'$, and for all $1 \leq i \leq k$, we have $\lambda(U_i) = V_i$.
The map is lifted to sets of bounded summaries in the natural way. We also lift the map *Unbound* to bounded environments. For a bounded environment $\mathcal{E}_{BS}$, the set $Unbound(\mathcal{E}_{BS})$ is the environment $\mathcal{E}$ in $\mathcal{T}^V(\mathcal{M})$ such that for every variable X, we have $\mathcal{E}(X) = Unbound(\mathcal{E}_{BS}(X))$.
Now observe that:

1. For any bounded summary $\mathbf{b}$, the set $Unbound(\mathbf{b})$ is $\equiv_{\mathcal{M}}$-closed.

2. For any bounded environment $\mathcal{E}_{BS}$, the environment $Unbound(\mathcal{E}_{BS})$ is $\equiv$-closed.

Next we show inductively that:

Lemma 3. *For any* NT-μ *formula* φ*, bounded environment* $\mathcal{E}_{BS}$*, and bounded summary* $\mathbf{b}$*, and for all* $\mathbf{s} \in Unbound(\mathbf{b})$*, we have*

$$\mathbf{b} \in Eval(\varphi, \mathcal{E}_{BS}) \textit{ iff } \mathbf{s} \in [\![\varphi]\!]_{Unbound(\mathcal{E}_{BS})}.$$

Now note that if initial state of $\mathcal{M}$ is v_{in} and the root of $\mathcal{T}(\mathcal{M})$ is s_{in}, then $\langle s_{in} \rangle \in Unbound(\langle v_{in} \rangle)$. Also note that $Unbound(\perp_B) = \perp$, where $\perp$ is the empty environment for $\mathcal{T}(\mathcal{M})$. From Lemma 3, it follows that:

Theorem 3. *For an NSM* $\mathcal{M}$ *with initial state* v_{in} *and a closed* NT-μ *formula* φ*,* $\mathcal{T}(\mathcal{M})$ *satisfies* φ *if and only if* $\langle v_{in} \rangle \in Eval(\varphi, \perp_B)$*. Further,* $Eval(\varphi, \perp_B)$ *is inductively computable.*

To understand this more concretely, let us see how this model-checking algorithm runs on a simple NSM. Consider the NSM abstraction $\mathcal{M}_{foo}$ in Sec. 2.2. The states of this NSM are v_1, v_2, v_3, v_4, v_5, and v_2'; the initial state is v_1. The states are labeled by the atomic propositions rd, wr, tk, en, and ex using a map defined as: $\kappa(v_0) = \emptyset$, $\kappa(v_1) = \{wr\}$, $\kappa(v_2) = \{en\}$, $\kappa(v_2') = \{ex\}$, $\kappa(v_3) = \{tk\}$, $\kappa(v_4) = \{rd\}$, and $\kappa(v_5) = \{end\}$. The transition relations of $\mathcal{M}_{foo}$ are given by:

1. $\Delta_{call} = \{(v_2, v_1)\}$,

2. $\Delta_{loc} = \{(v_1, v_2), (v_1, v_3), (v_2', v_4), (v_2', v_5), (v_3, v_4), (v_3, v_5), (v_4, v_4), (v_4, v_5)\}$, and

3. $\Delta_{ret} = \{(v_5, v_2, v_2')\}$.

Now suppose we want to check if a write action is locally reachable from the initial state. The NT-μ property specifying this requirement is $\varphi = \mu X.(wr \vee \langle loc\rangle X \vee \langle call\rangle \varphi'\{X\})$, where $\varphi' = \mu Y.(\langle ret\rangle R_1 \vee \langle loc\rangle Y \vee \langle call\rangle Y\{Y\})$.

We show how to compute the set of bounded summaries satisfying φ'—the computation for φ is very similar. After the first iteration of the fixpoint computation that builds this set, we obtain the set $\mathbf{S_1} = \{\{\langle v_5, v_2, \{v_2'\}\rangle\}$ (the set of summaries satisfying $\langle ret\rangle R_1$). After the second step, we obtain the set of summaries $\mathbf{S_2} = \mathbf{S_1} \cup \{\langle v_2', v_2, \{v_2'\}\rangle, \langle v_3, v_2, \{v_2'\}\rangle, \langle v_4, v_2, \{v_2'\}\rangle\}$, and the next set computed is $\mathbf{S_3} = \mathbf{S_2} \cup \{\langle v_1, v_2, \{v_2'\}\rangle\}$. Note that in these two steps, we only use local edges in the NSM. Now, however, we have found a bounded summary starting at the "entry state" of the procedure *foo*, which may be plugged into the recursive call to *foo*. More precisely, we have $(v_2, v_1) \in \Delta_{call}$, $\langle v_1, v_2, \{v_2'\}\rangle \in \mathbf{S_3}$, and $\langle v_2', v_2, \{v_2'\}\rangle \in \mathbf{S_3}$, so that we may now construct $\mathbf{S_4} = \mathbf{S_3} \cup \langle v_2, v_2, \{v_2'\}\rangle$. This ends the fixpoint computation, so that $\mathbf{S_4}$ is the set of summaries satisfying φ'.

Let us now analyze the complexity of this algorithm. Let N_V be the number of states in $\mathcal{M}$, and let n be the arity of the formula in question. Then the total number of bounded summaries in $\mathcal{M}$ that we need to consider is bounded by $N = N_V^2 2^{N_V n}$. Let us now assume that union or intersection of two sets of summaries, as well as membership queries on such sets, take linear time. It is easy to see that the time needed to evaluate a non-fixpoint formula φ of arity $n \leq |\varphi|$ is bounded by $O(N^2|\varphi|N_V)$ (the most expensive modality is $\langle call\rangle\varphi'\{\psi_1, \ldots, \psi_n\}$, where we have to match an "inner" summary satisfying φ' as well as n "outer" summaries satisfying the ψ_i-s). For a fixpoint formula φ with one fixpoint variable, we may need N such evaluations, so that the total time required to evaluate $Eval(\varphi, \perp_B)$ is $O(N^3|\varphi|N_V)$. For a formula φ of alternation depth d, this evaluation takes time $O(N^{3d}N_V^d\ |\varphi|)$, i.e., exponential in the sizes of $\mathcal{M}$ as well as φ.

It is known that model-checking alternating reachability specifications on a pushdown model is EXPTIME-hard [Wal01]. Following constructions similar to those in Section 3.3, we can generate a NT-μ formula φ from a μ-calculus formula f expressing an alternating reachability property such that: (1) the size of φ is linear in the size of f, and (2) $\mathcal{M}$ satisfies φ if and only if $\mathcal{M}$ satisfies f. It follows that model-checking a closed NT-μ formula φ on an NSM $\mathcal{M}$ is EXPTIME-hard. Combining, we conclude that:

Theorem 4. *Model checking a* NT-μ *formula* φ *on an NSM* $\mathcal{M}$ *is* EXPTIME-*complete.*

Better bounds may be obtained if the formula has a certain restricted form. For instance, it can be shown that for linear time (Büchi or reachability) requirements, model-checking takes time polynomial in the number of states of $\mathcal{M}$. The reason is that in this case, it suffices to only consider bounded summaries of the form $\langle v, v', \{v''\}\rangle$, which are polynomial in number. The fixpoint computation stays the same.

Note that our decision procedure is very different from known methods for branching-time model-checking of pushdown models [Wal01, BS99]. The latter are complex and seem hard to implement; our algorithm, being symbolic in nature, seems to be a step in the direction of practicality. An open question here is how to represent sets of bounded summaries symbolically. Also, note that our algorithm directly implements the operational semantics of NT-μ formulas over bounded summaries. In this regard NT-μ resembles the modal μ-calculus, whose formulas encode fixpoint computations over sets; to model-check μ-calculus formulas, we merely need to perform these

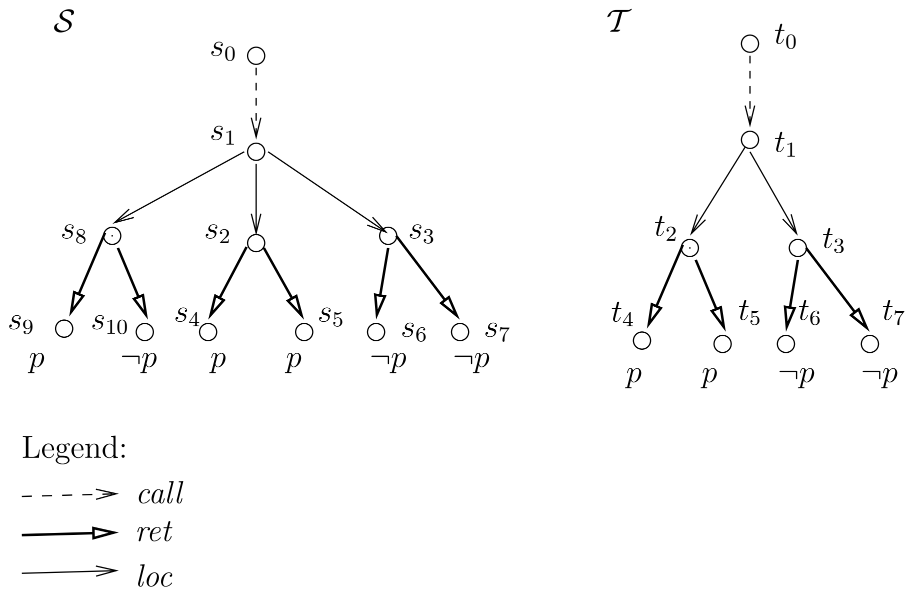

Figure 3.7: An arity hierarchy

computations. Unsurprisingly, our procedure is very similar to classical symbolic model-checking for the μ-calculus.

3.5 A hierarchy theorem

We will study the expressiveness of NT-μ in detail in Chapter 4, when we examine its connections with automata on nested trees and existing temporal logics like the μ-calculus and CARET. In this section, we show that the expressiveness of NT-μ formulas increases with their arity.

For two nested trees $\mathcal{T}_1$ and $\mathcal{T}_2$ respectively rooted at nodes s_1 and s_2, we say $\mathcal{T}_1$ and $\mathcal{T}_2$ are *distinguished* by a closed, k-ary NT-μ formula φ if and only if s_1 satisfies φ and s_2 does not. Then we have:

Theorem 5. *For every $k \geq 1$, there is a a closed $(k+1)$-ary formula φ_{k+1}, and two nested trees $\mathcal{T}_1$ and $\mathcal{T}_2$, such that φ_{k+1} can distinguish between $\mathcal{T}_1$ and $\mathcal{T}_2$, but no closed k-ary* NT-μ *formula can.*

We will sketch the proof for the case $k = 1$. Before we do so, we need some extra machinery. More precisely, we will define a preorder called *quasi-bisimilarity* over summaries that takes into account their coloring. It turns out that NT-μ respects this preorder.

Consider a pair of k-colored summaries $\mathbf{s} = \langle s, U_1, \ldots, U_k \rangle$ and $\mathbf{t} = \langle t, V_1, \ldots, V_k \rangle$ such that each path in the trees rooted at s and t comprises a chain of *loc*-edges followed by one *ret*-edge leading to a leaf. Let S and T respectively be the sets of non-leaf nodes in these trees. We say that $\mathbf{s}$ and $\mathbf{t}$ are *quasi-bisimilar* if there is a relation $\preceq \subseteq S \times T$ such that $s \preceq t$ and

1. For all $s' \preceq t'$, we have $\lambda(s') = \lambda(t')$
2. If $s' \preceq t'$, then for every s'' such that $s' \xrightarrow{loc} s''$, there is a t'' such that $t' \xrightarrow{loc} t''$ and $s'' \preceq t''$. Also, for every t'' such that $t' \xrightarrow{loc} t''$, there is an s'' such that $s' \xrightarrow{loc} s''$ and $s'' \preceq t''$.
3. If $s' \preceq t'$, then for every s'' such that $s' \xrightarrow{ret} s''$, there is a t'' such that $t' \xrightarrow{ret} t''$, and for every t'' such that $t' \xrightarrow{ret} t''$, there is an s'' such that $s' \xrightarrow{ret} s''$. Further, if $s'' \in U_i$ then $t'' \in V_i$, for all i (note that this is not an "iff" condition).

Now, we can show inductively that if $\mathbf{s}$ and $\mathbf{t}$ are quasi-bisimilar, then for every variable-free NT-μ formula φ, if $\mathbf{s}$ satisfies φ, then $\mathbf{t}$ satisfies φ as well (note that the converse is not true; for instance, $\mathbf{t}$ may satisfy $[ret]R_i$ even when $\mathbf{s}$ does not). We skip the proof.

Let us now come back to Theorem 5. Consider the two non-bisimilar nested trees $\mathcal{S}$ and $\mathcal{T}$ in Figure 3.7 with initial nodes s_0 and t_0 (the jump-edges are not shown, but can be reconstructed from the edge labeling). It is easy to see that the 2-ary NT-μ formula $\varphi = \langle call \rangle(\langle loc \rangle(\langle ret \rangle R_1 \wedge \langle ret \rangle R_2))\{p, \neg p\}$ distinguishes s_0 and t_0. Let us now see if there is a closed, 1-ary formula φ that can distinguish between $\mathcal{S}$ and $\mathcal{T}$. First, if φ is a disjunction or conjunction, we can get a smaller witness for this distinction. Further, because trees $\mathcal{S}$ and $\mathcal{T}$ are of fixed depth, we need only consider fixpoint-free formulas. The interesting case is that of formulas of the form $\varphi = \langle call \rangle \varphi' \{\psi\}$.

Assume this formula is satisfied by $\langle s_0 \rangle$; then there is a bisimulation-closed summary of the form $\mathbf{s} = \langle s_1, U \rangle$ that satisfies φ'. For each such $\mathbf{s}$, we find a $\mathbf{t} = \langle t_1, V \rangle$. Note that $\mathbf{s}$ can assume only four values; these are $\langle s_1, \{s_9, s_4, s_5\} \rangle$, $\langle s_1, \{s_{10}, s_6, s_7\} \rangle$, $\langle s_1, \{s_9, s_4, s_5, s_{10}, s_6, s_7\} \rangle$, and $\langle s_1, \emptyset \rangle$. The corresponding values of $\mathbf{t}$ are $\langle t_1, \{t_6, t_7\} \rangle$, $\langle t_1, \{t_4, t_5\} \rangle$, $\langle t_1, \{t_4, t_5, t_6, t_7\} \rangle$, and $\langle t_1, \emptyset \rangle$ respectively. Note that for any value $\mathbf{s}$ takes, the corresponding $\mathbf{t}$ is quasi-bisimilar to it, which means that $\mathbf{t}$ satisfies φ'. Further, for each $v \in V$ there is a bisimilar node $u \in U$. It follows that if all $u \in U$ satisfy ψ, then so do all $v \in V$. Then $\langle t_0 \rangle$ satisfies φ.

Similarly one can show that $\langle t_0 \rangle$ satisfies φ only if $\langle s_0 \rangle$ satisfies φ.

To extend the proof to arbitrary k, we consider a structure $\mathcal{S}'$ where, like in $\mathcal{S}$, the root has one *call*-child s_1—except s_1 now has a large number N of *loc*-children s'. From each s', we have $(k+1)$ *ret*-edges leading to "leaves" s'', each of which is labeled with exactly one proposition from the set $AP = \{p_1, p_2, \ldots, p_{k+1}\}$. For $(N-1)$ values of s', the leaves of the trees rooted at s' are labeled such that only k of them have distinct labels. But there is a particular s' (call it s'_d) for which these leaves get distinct labels $p_1, \ldots, p_{k+1}$.

Now take a structure $\mathcal{T}'$ that is obtained by removing the subtree rooted at node s'_d from $\mathcal{S}'$. Following the methods for the case $k = 1$, we can show that $\mathcal{S}'$ and $\mathcal{T}'$ may be distinguished by a $(k+1)$-ary formula, but by no k-ary formula. We skip the details.

Chapter 4

Automata and monadic second-order logic on nested trees

In this chapter, we study finite automata and monadic second-order logic on nested trees. The former can send states down a nested tree along tree edges as well as jump-edges; the latter can explicitly refer to jump-edges via a special predicate. In addition to studying the properties of these formalisms independently, we also identify connections between them and relate them to NT-μ. The main technical result is that NT-μ and alternating parity automata on nested trees have the same expressive power. This result is the analog of the equivalence between the modal μ-calculus and alternating parity automata on trees [EJ91].

The chapter is organized as follows. In Section 4.1, we define nested tree automata in their nondeterministic and alternating flavors and compare their closure properties and expressiveness. Section 4.2 proves that NT-μ is expressively equivalent to AP-NTAs. In Section 4.3, we introduce and study MSO-logic over nested trees.

4.1 Automata on nested trees

In this section, we study finite-state automata operating on nested trees. Recall that for tree automata, the state while reading a (non-root) tree node depends on its state at the node's parent. The state of a *nested tree automaton* (NTA) at a node in a nested tree depends on its states at the node's parent and the node's jump-predecessor (if it exists). We define these automata in nondeterministic and alternating flavors; the natural semantics of these are respectively over ordered and unordered nested trees. Regarding acceptance conditions, we focus on the parity condition for the most part as it is the most powerful among the popular ω-acceptance conditions; however, we also discuss acceptance by final state.

We start with nondeterministic automata. Formally, a (top-down) *nondeterministic parity nested tree automaton* (NP-NTA) over Σ is a structure $\mathcal{A} = (Q, q_0, \Delta, \Omega)$ where Q is a finite set of *states*, $q_0 \in Q$ is the initial state, $\Delta \subseteq Q \times \Sigma \times (TT \times TT)$, where $TT = Q \cup (Q \times Q) \cup \{\perp\}$, is a *transition relation*, and $\Omega : Q \to \{0, 1, \ldots, n\}$, for some $n \in \mathbb{N}$, is the *parity accepting condition* that assigns a *priority* to each automaton state.

NP-NTAs accept languages of ordered, binary nested trees. A run of $\mathcal{A}$ on an ordered, binary nested tree $\mathcal{T} = ((S, r, \to_1, \to_2), \hookrightarrow, \lambda)$ is a labeling $\rho : S \to Q$ of nodes of $\mathcal{T}$ by automaton states such that:

1. $\rho(r) = q_0$;

2. for all s, if $\rho(s) = q$ and $\lambda(s) = \sigma$, then

   - if s_1 and s_2 are the left and right children of s, then for some $(q, \sigma, (\tau_1, \tau_2)) \in \Delta$, we have: (a) if s_i, for $i \in \{1, 2\}$, is a call or local node, then $\tau_i = \rho(s_i)$, and (b) if s_i is a return node, then $\tau_i = (\rho(t), \rho(s_i))$, where $t \hookrightarrow s_i$

   - if s' is the left (similarly, right) child of s and s has no right (similarly, left) child, then for some $(q, \sigma, (\tau', \perp)) \in \Delta$ (similarly, $(q, \sigma, (\perp, \tau')) \in \Delta$),

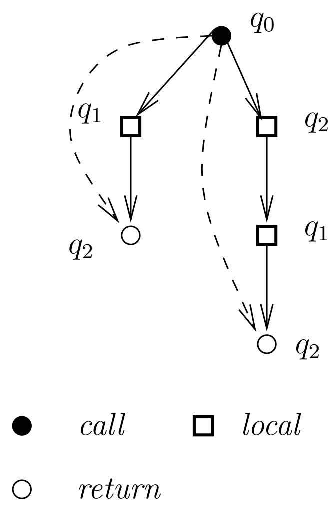

Figure 4.1: A run of a nondeterministic NTA

we have: (a) if s' is a call or local node, then $\tau' = \rho(s')$, and (b) if s' is a return node, then $\tau' = (\rho(t), \rho(s'))$, where $t \hookrightarrow s'$

Let π_i denote the i-th vertex in a path π in $\mathcal{T}$. A run ρ of $\mathcal{A}$ on $\mathcal{T}$ is *accepting* if for all infinite paths π in $\mathcal{T}$, $\theta' = \max\{\theta : \Omega(\rho(\pi_i)) = \theta$ for infinitely many $i\}$ is even. An ordered, binary nested tree $\mathcal{T}$ is *accepted* if $\mathcal{A}$ has an accepting run on it. The *language* $\mathcal{L}(\mathcal{A})$ of $\mathcal{A}$ is the set of nested trees it accepts.

Figure 4.1 illustrates part of a run of an NP-NTA on a nested tree (assume that the label of every node is σ and that every node has a left child). Transitions include $(q_1, \sigma, ((q_0, q_2), \perp))$ and $(q_0, \sigma, (q_1, q_2))$.

Note that we can define an equivalent semantics of NP-NTAs by letting the automaton manipulate a stack rather than consult a node's jump-predecessor. In this case, $\mathcal{A}$ pushes the current state while taking a call edge, pops on a return edge, and leaves the stack unchanged on a local edge. As jump-edges are nested, the top of the automaton stack at a return node stores the state at the node's matching call.

As we shall see, unlike their analogs on trees, nondeterministic automata on nested trees do not have robust closure properties. However, this problem goes away for *alternating* nested tree automata, which we will now consider. We will interpret these automata on unordered nested trees as these are the structures generated from our program abstractions—they are more naturally interpreted on unordered nested trees anyway. Also, their semantics are defined more easily if we let them manipulate stacks of states.

Formally, for a finite set Q, define the set $TT(Q)$ of *transition terms* whose members f are of the form $f := tt \mid ff \mid f \vee f \mid f \wedge f \mid \langle loc \rangle q \mid [loc]q \mid \langle call \rangle q \mid [call]q \mid \langle ret, q' \rangle q \mid [ret, q']q$, where $q, q' \in Q$. An *alternating parity nested tree automaton* (AP-NTA) over Σ is a structure $\mathcal{A} = (Q, q_0, \Delta, \Omega)$, where Q is a finite set of states, $q_0 \in Q$ is the initial state, $\Delta : Q \times \Sigma \to TT(Q)$ is a transition function, and $\Omega : Q \to \{0, 1, \ldots, n\}$ is the parity accepting condition.

We define the semantics of an AP-NTA $\mathcal{A} = (Q, q_0, \Delta, \Omega)$ via a parity game. The acceptance game $\mathcal{G}(\mathcal{A}, \mathcal{T})$ of a Σ-labeled nested tree $\mathcal{T} = (T, \hookrightarrow, \lambda)$ by $\mathcal{A}$ is played by two players A and E. The vertex set of the game graph is $\mathcal{V} = T \times Q \times Q^* \times TT$, and the set of moves $\Rightarrow \subseteq \mathcal{V} \times \mathcal{V}$ is the least set such that:

- for all $v \in \mathcal{V}$ of the form $(s, q, \alpha, f_1 \vee f_2)$ or $(s, q, \alpha, f_1 \wedge f_2)$ for some $v' \in \mathcal{V} \cup \{\epsilon\}$, we have $v \Rightarrow (s, q, \alpha, f_1)$ and $v \Rightarrow (s, q, \alpha, f_2)$;
- for all $v \in \mathcal{V}$ of the form $(s, q, \alpha, \langle loc \rangle q'))$ or $(s, q, \alpha, [loc]q'))$, and for all s' such that $s \xrightarrow{loc} s'$, we have $v \Rightarrow (s', q', \alpha, f)$, where $f = \Delta(q', \lambda(s'))$;
- for all $v \in \mathcal{V}$ of the form $(s, q, \alpha, \langle call \rangle q')$ or $(s, q, \alpha, [call]q')$, and for all s' such that $s \xrightarrow{call} s'$, we have $v \Rightarrow (s', q', q.\alpha, f)$, where $f = \Delta(q', \lambda(s'))$;
- for all $v \in \mathcal{V}$ of the form $(s, q, q''.\alpha, \langle ret, q'' \rangle q')$ or $(s, q, q''.\alpha, [ret, q'']q')$, and for all s' such that $s \xrightarrow{ret} s'$, we have $v \Rightarrow (s', q', \alpha, f)$, where $f = \Delta(q', \lambda(s'))$;

The vertex set $\mathcal{V}$ is partitioned into two sets $\mathcal{V}_{\mathsf{E}}$ and $\mathcal{V}_{\mathsf{A}}$ corresponding to the two players. The set $\mathcal{V}_{\mathsf{A}}$ comprises vertices of the form (s, q, α, f), where s, q and α are arbitrary and f has the form tt, $[call]q$, $[loc]q$, $[ret, q']q$, or $(f_1 \wedge f_2)$. The remaining vertices constitute $\mathcal{V}_{\mathsf{E}}$. We also lift the priority map Ω to $\Omega_{\mathcal{V}} : \mathcal{V} \to \{0, 1, \ldots, n\}$ by defining $\Omega_{\mathcal{V}}(s, q, \alpha, f) = \Omega(q)$ for all s, q, α, and f.

The two players A and E play on the graph starting from the initial position $v_{in} = (s_0, q_0, \epsilon, \Delta(q_0, \lambda(s_0)))$ by moving a token along edges of the game graph. Whenever the token is in a position v, the player who owns the vertex must move the token. Formally, a *play* of $\mathcal{G}$ is a non-empty, finite or infinite sequence $\alpha = v_1 v_2 \ldots$ that is a path in the game graph, where $v_1 = v_{in}$. A finite play is winning for player A if the last position is a player E vertex from which there is no move; analogously, we define winning finite plays for player E. An infinite play α is winning for player E if $\theta' = \max\{\theta : \Omega_{\mathcal{V}}(v_i) = \theta \text{ for infinitely many } i\}$ is even; otherwise A wins the play (this is the standard *max-parity* acceptance condition for parity games). A *strategy* for player E (or A) is a subset of edges $\mathit{Str} \subseteq \Rightarrow$ such that all these edges originate in a vertex in $\mathcal{V}_{\mathsf{E}}$ (or $\mathcal{V}_{\mathsf{A}}$)[1]. A play is according to a strategy Str if all edges in the play are in Str. A strategy is winning if all maximal plays according to the strategy are winning.

An AP-NTA $\mathcal{A}$ *accepts* a nested tree $\mathcal{T}$ if E has a winning strategy in $G(\mathcal{A}, \mathcal{T})$. The *language* $\mathcal{L}(\mathcal{A})$ of $\mathcal{A}$ is the set of nested trees accepted by $\mathcal{A}$.

We also consider automata that accept by the weaker *final-state condition*. For nondeterministic versions of such automata, a nested tree is accepted if a special *final state* q_f is seen along every path in some run on it. In alternating versions, all infinite plays are won by A, and if a play reaches a game vertex (s, q_f, α, f) for some s, α, and f, then the game terminates and E is the winner.

4.1.1 Closure properties

We start off with some positive closure properties of AP-NTAs and NP-NTAs:

[1] Strategies are often defined in a more general way that refer to the history of the play. This definition suffices as parity games always admit zero-memory strategies [GTW02].

Theorem 6. *AP-NTAs are closed under union, intersection, and complement. NP-NTAs are closed under union and intersection.*

Proof. Proofs that AP-NTAs are closed under union and intersection are easy, and mirror corresponding results for alternating tree automata. We give a proof sketch of the closure of AP-NTAs under complement.

Consider an AP-NTA$\mathcal{A} = (Q, q_0, \Delta, \Omega)$. Let us define, for every transition term f, the transition term $Neg(f)$ as follows:

$$\begin{aligned}
Neg(f_1 \vee f_2) &= Neg(f_1) \wedge Neg(f_2) \\
Neg(f_1 \wedge f_2) &= Neg(f_1) \vee Neg(f_2) \\
Neg(tt) &= ff \\
Neg(ff) &= tt \\
Neg(\langle loc \rangle q) &= [loc]q \\
Neg([loc]q) &= \langle loc \rangle q \\
Neg(\langle call \rangle q) &= [call]q \\
Neg(()[call]q) &= \langle call \rangle q \\
Neg(\langle ret, q' \rangle q) &= [ret, q']q \\
Neg([ret, q']q) &= \langle ret, q' \rangle q
\end{aligned}$$

Let $\Delta_\neg$ be the transition function defined as: if $\Delta(q, \sigma) = f$ for some q, σ, f, then $\Delta_\neg(q, \sigma) = Neg(f)$. Also, we define a new parity acceptance condition $\Omega_\neg$ as: for all $q \in Q$, $\Omega_\neg(q) = 1 + \Omega(q)$. Now we construct an AP-NTA $\mathcal{A}_\neg$ which is the same as $\mathcal{A}$, except its transition function is $\Delta_\neg$ and acceptance condition $\Omega_\neg$.

Now suppose that in the acceptance game $\mathcal{G}$ for $\mathcal{A}$ and a nested tree $\mathcal{T}$, Player E has a winning strategy (i.e., $\mathcal{T}$ is not accepted by $\mathcal{A}$). It is known, from research on parity games, that this strategy is *memoryless* [GTW02]. It can now be shown that the same strategy, applied to the acceptance game for $\mathcal{T}$ and $\mathcal{A}_neg$, is winning for Player A. As for the other direction, suppose Player E has no winning strategy in $\mathcal{G}$.

As parity games are *determined*, this means A has a memoryless winning strategy in $\mathcal{G}$. We can now show that the same strategy is winning for E in the acceptance game for $\mathcal{A}_\neg$ (the arguments are the same as in the proof of closure of alternating tree automata under complement [GTW02]— as a result, we leave the details out). It follows that $\mathcal{A}_\neg$ accepts the complement of $\mathcal{L}(\mathcal{A})$.

Regarding NP-NTAs, their closure under union is trivial. As for closure under intersection, consider NP-NTAs $\mathcal{A}_1 = (Q^1, q_0^1, \Delta^1, \Omega^1)$ and $\mathcal{A}_2 = (Q^2, q_0^2, \Delta^2, \Omega^2)$, and construct an NP-NTA $\mathcal{A}_\cap = (Q, q_0, \Delta, \Omega)$. Here, $Q = Q^1 \times Q^2$ and $q_0 = (q_0^1, q_0^2)$. Let us now define the "product" $\mathit{Prod}(\tau_1, \tau_2)$ of two transition terms τ_1 and τ_2 as follows:

- if $\tau_1 = \perp$ and $\tau_2 = \perp$, then $\mathit{Prod}(\tau_1, \tau_2) = \perp$;
- if $\tau_1 = q_1$ and $\tau_2 = q_2$, then $\mathit{Prod}(\tau_1, \tau_2) = (q_1, q_2)$;
- if $\tau_1 = (q_1', q_1)$ and $\tau_2 = (q_2', q_2)$, then $\mathit{Prod}(\tau_1, \tau_2) = ((q_1', q_2'), (q_1, q_2))$.

Then the transition relation Δ is the least relation such that for each transition $(q^1, a, (\tau_1^1, \tau_2^1))$ in Δ^1 and $(q^2, a, (\tau_1^2, \tau_2^2))$ in Δ_2 such that $\mathit{Prod}(\tau_1^1, \tau_1^2)$ and $\mathit{Prod}(\tau_2^2, \tau_2^2)$ are defined, we have a transition $((q^1, q^2), a, (\mathit{Prod}(\tau_1^1, \tau_1^2), \mathit{Prod}(\tau_2^2, \tau_2^2)))$ in Δ. Finally, we define Ω such that for all $(q^1, q^2) \in Q^1 \times Q^2$, we have $\Omega((q^1, q^2)) = \max\{\Omega^1(q^1), \Omega^2(q^2)\}$.

It can now be verified that $\mathcal{L}(\mathcal{A}_\cap) = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$.

□

Observe that by our definition, languages accepted by AP-NTAs are closed under bisimulation, while those accepted by NP-NTAs are not in general. To compare the expressiveness of an AP-NTA and an NP-NTA meaningfully, we need to consider the language obtained by starting with the language L of the NP-NTA, stripping the order between tree edges off nested trees in L, and closing it under bisimulation.[2]

[2]Alternatively, we could define AP-NTAs on ordered nested trees. Under this definition as well, AP-NTAs are strictly more powerful than NP-NTAs.

Formally, for a language L of ordered nested trees, we define $Unord(L)$ as the bisimulation closure of the set of nested trees $((S, r, \rightarrow), \hookrightarrow, \lambda)$ such that $\rightarrow = \rightarrow_1 \cup \rightarrow_2$ for some $((S, r, \rightarrow_1, \rightarrow_2), \hookrightarrow, \lambda) \in L$. Now we show that:

Theorem 7. *There is a language L of ordered, binary nested trees such that: (1) there is no NP-NTA accepting L, and (2) there is an AP-NTA accepting $Unord(L)$.*

Proof. Consider ordered nested trees of the form in Figure 4.2-a, where $\Sigma = \{0, 1, \$\}$, and $a_i, b_{ij} \in \Sigma$ for all i, j (while the structure in the figure is not binary, it can be encoded as such; also, the jump-edges, omitted to keep the figure clean, can be reconstructed). Let L_{gap} be the language of such structures where for all $i \leq n$, there is some $k \leq m$ such that $a_{n-i+1} = b_{ki}$. First, we note that L_{gap} cannot be recognized by an NP-NTA $\mathcal{A}_N$ with N states. To see why, take a structure as above where $n = m > N$, and for each $1 \leq i \leq n$, there is a *distinct* branch k such that $a_{n-i+1} = b_{ki}$. In any run, $\mathcal{A}_N$ must enter two branches in the same state; also, the sequence of states at calls unmatched till these points are the same. We can replace one of these branches with the other to get an accepting run on a structure not in L_{gap}.

On the other hand, it is easy to build an AP-NTA $\mathcal{A}$ that recognizes $Unord(L_{gap})$. The automaton has a state q, q_0^* and q_1^*. While reading the sequence of symbols a_i, the automaton uses alternation to assert a condition at each i (we can view this as the automaton "forking" a copy). This is done such that the copy of $\mathcal{A}$ forked at the node labeled a_{n-i+1} has state $q^*_{a_{n-i+1}}$ at that node. On reading the next symbol, this copy changes its state to q; it continues reading the structure in this state till it reaches the node marked \$, at which point it nondeterministically chooses to "check" branch k. When it reaches the node labeled b_{ki} (i.e., the node s such that at the node t satisfying $t \hookrightarrow s$, the automaton had state q_j^* for some j), it can check if $b_{ki} = a_{n-i+1}$ and accept the nested tree. $\mathcal{A}$ accepts the structure iff every "copy" forked this way accepts it. $\square$

The above is an example of how automata theory for nested trees differs from the

theory of tree automata. In the latter setting, allowing alternation does not affect an automaton's expressiveness. In fact, this is also a point of difference between nested tree automata and *automata nested words* [AM04, AM06], as alternating and nondeterministic automata on nested words are expressively equivalent.
Further, Theorem 7 implies that:

Theorem 8. *NP-NTAs are not closed under complementation.*

Proof. We give an NP-NTA $\mathcal{A}_N$ that accepts the complement of the language L_{gap} in the proof for Theorem 7. Consider a nested tree $\mathcal{T}$ in this language. We can assume that $\mathcal{T}$ has the same "shape" as the nested tree in Figure 4.2-(a)— i.e., there is a chain of nodes from the root that ends with a node with multiple children, each of which leads to a chain of nodes. This is because an NP-NTA can determine if a nested tree violates this property, and accept it if it does.
While running on nested trees of this form, $\mathcal{A}_N$ nondeterministically guesses the i such that a_i cannot be "matched" along any of the branches, and lets its state at the node labeled a_i be $q^*_{a_i}$ (at other nodes along the stem of a-s, its state is q). Now it sends a state to each branch to determine a node s such that the automaton state at the node t, where $t \hookrightarrow s$, is of the form q^*_j. It compares the label b_{jk} of this node with a_i, and rejects the nested tree if $b_{jk} = a_i$. The accepts if all the copies sent along the different branches accept. It can now be easily verified that $\mathcal{A}_N$ accepts the complement of L_{gap}. ☐

The projection over Σ_1 of a language L of (ordered, unordered) nested trees over $\Sigma_1 \times \Sigma_2$ is the language $Proj_{\Sigma_1}(L)$ obtained by replacing every label (a, b) in every nested tree $\mathcal{T} \in L$ by a. We can show that:

Theorem 9. *NP-NTAs are closed under projection, but AP-NTAs are not.*

Proof. For NP-NTAs, closure under projection is easy. Consider nested trees over $\Sigma_1 \times \Sigma_2$ as above, and let there be an NP-NTA $\mathcal{A}$ accepting a language of such trees. An NP-NTA $\mathcal{A}'$ accepting the language $Proj_{\Sigma_1}(L)$ is constructed as follows. While reading a node labeled $a \in \Sigma_1$, $\mathcal{A}'$ guesses a second component $b \in \Sigma_2$ using nondeterminism, and mimics a move of $\mathcal{A}$ on the label (a, b). It is easy to see that $\mathcal{A}'$ accepts $Proj_{\Sigma_1}(L)$.

We postpone the argument for the second part of the theorem to Section 4.3. $\square$

4.1.2 Decision problems

The *model checking* problem for AP-NTAs on nested state machines is the problem of deciding, given an AP-NTA $\mathcal{A}$ and a nested state machine $\mathcal{M}$, whether $\mathcal{T}(\mathcal{M}) \in \mathcal{L}(\mathcal{A})$.

Theorem 10. *The model-checking problem for AP-NTAs on nested state machines is* EXPTIME*-complete.*

Proof. We obtain an EXPTIME procedure for this problem via a reduction to a *pushdown parity game*. A two-player pushdown parity game is a parity game played on the configuration graph of a pushdown system. It is known that pushdown parity games are solvable in EXPTIME [Wal01]. Now, given an AP-NTA $\mathcal{A}$ and a nested state machine $\mathcal{M}$, $\mathcal{T}(\mathcal{M}) \in \mathcal{L}(\mathcal{A})$ iff player E wins the acceptance game of $\mathcal{A}$. Now recall that call-edges (similarly, return-edges) in $\mathcal{T}(\mathcal{M})$ encode call transitions (similarly, return transitions) of $\mathcal{M}$— i.e., these capture pushes (similarly pops) to the stack implicit in an NSM abstraction. However, these edges are also where the stack of states in the semantics of $\mathcal{A}$ is pushed (popped). Thus, the implicit stack of $\mathcal{M}$ is "synchronized" with the implicit stack of $\mathcal{A}$, so that the graph of the acceptance game of $\mathcal{T}(\mathcal{M})$ by $\mathcal{A}$ happens to be the configuration graph of a pushdown system that is roughly the "synchronized product" of $\mathcal{M}$ and $\mathcal{A}$.

An EXPTIME-hardness result for this problem follows from the known hardness of the model-checking problem for alternating tree automata on pushdown systems [Wal01]. $\square$

While model checking for alternating NTAs is decidable, emptiness is not[3]:

Theorem 11. *Universality for nondeterministic NTAs and emptiness for alternating NTAs are undecidable problems, even for acceptance by final state.*

Proof. The proof employs a reduction from the Post's Correspondence Problem (PCP) [HU79]. Consider a tuple $((u_1, \ldots, u_k), (v_1, \ldots, v_k))$, where the u_i's and v_i's are finite words over an alphabet A; the PCP is to determine if there is a sequence $i_1, \ldots, i_m$, where $i_j \leq k$, such that $u_{i_1} u_{i_2} \ldots u_{i_m} = v_{i_1} v_{i_2} \ldots v_{i_m} = w$. Now consider nested trees of the form in Figure 4.2-(b) (the jump-edges are omitted to keep the figure simple) such that the initial call-chain is of length m and is labeled by symbols from the alphabet $\{1, \ldots, k\}$, and the symbols w_i on the "stem" of local nodes succeeding this chain form the string w. Now suppose the sequence of input symbols on the call chain is $c_{i_m} \ldots c_{i_1}$. There are two kinds of return chains hanging from the stem—the ones marked with the symbol $*$ (similarly \$) are exactly at the points where w may be possibly factored into $u_{i_1}, u_{i_2}, \ldots, u_{i_m}$ (similarly $v_{i_1}, \ldots, v_{i_m}$). Also, the i-th return chain (counting from left) of either type is of length i. Then such a nested tree is a witness for an instance of PCP being positive. We can, however, show that there is an alternating NTA accepting by final state that accepts the set of nested trees bisimilar to such witnesses. In fact, we can show that there is a nondeterministic final-state NTA that accepts any nested tree *not* of the above form (under some ordering of edges). □

However, we can prove the emptiness problem of NP-NTAs to be solvable in EXPTIME by reducing it to that for pushdown tree automata [KPV02].

4.2 Relation between NT-μ and NTAs

We now establish our main theorems, which show that AP-NTAs are exactly as expressive as NT-μ.

[3] This result was obtained independently by Löding [Lö5].

Theorem 12. *Given any closed* NT-μ *formula* φ*, we can construct an AP-NTA* $\mathcal{A}_\varphi$ *such that for any nested tree* $\mathcal{T}$*,* $\mathcal{T} \in \mathcal{L}(\varphi)$ *iff* $\mathcal{T} \in \mathcal{L}(\mathcal{A}_\varphi)$*. The size of* $\mathcal{A}_\varphi$ *is polynomial in the size of* φ*.*

Proof. The proof is similar in spirit to the known translation from the μ-calculus to alternating tree automata [EJ91]. The AP-NTA $\mathcal{A}_\varphi$ is over an input alphabet 2^{AP}. For every subformula ψ of φ, $\mathcal{A}_\varphi$ has a state q_ψ. The initial state is q_φ.
For any variable X in φ, let $\Psi(X)$ be the subformula of the form $\mu X.\varphi'$ or $\nu X.\varphi'$ that binds X (we assume that each variable in φ is bound at most once). For instance, if $\varphi = \langle call \rangle(\mu X.(p \vee X))\{q\}$, then $\Psi(X) = \mu X.(p \vee X)$. For each bound variable X in φ, the state q_X is *identified* with the state $q_{\Psi(X)}$.
Let $p \in AP$, and $\sigma \in 2^{AP}$. The transition relation Δ of $\mathcal{A}_\varphi$ is defined inductively over the structure of φ:

$$
\begin{aligned}
\Delta(q_p, \sigma) &= tt \text{ if } p \in \sigma, \text{ else } f\!f \\
\Delta(q_{\varphi_1 \wedge \varphi_2}, \sigma) &= \Delta(q_{\varphi_1}, \sigma) \wedge \Delta(q_{\varphi_2}, \sigma) \\
\Delta(q_{\varphi_1 \vee \varphi_2}, \sigma) &= \Delta(q_{\varphi_1}, \sigma) \vee \Delta(q_{\varphi_2}, \sigma) \\
\Delta(q_{\mu X.\varphi'}, \sigma) &= \Delta(q_{\varphi'}, \sigma) \\
\Delta(q_{\nu X.\varphi'}, \sigma) &= \Delta(q_{\varphi'}, \sigma) \\
\Delta(q_{\langle call \rangle(\varphi')\{\psi_1,\dots,\psi_k\}}, \sigma) &= \langle call \rangle q_{\varphi'} \\
\Delta(q_{[call](\varphi')\{\psi_1,\dots,\psi_k\}}, \sigma) &= [call] q_{\varphi'} \\
\Delta(q_{\langle loc \rangle \varphi'}, \sigma) &= \langle loc \rangle q_{\varphi'} \\
\Delta(q_{[loc]\varphi'}, \sigma) &= [loc] q_{\varphi'} \\
\Delta(q_{\langle ret \rangle R_i}, \sigma) &= \vee_{\phi', \psi_{1 \le j \le k}} (\langle ret, q_{\langle call \rangle(\phi')\{\psi_1,\dots,\psi_k\}} \rangle q_{\psi_i} \vee \\
&\qquad \langle ret, q_{[call](\phi')\{\psi_1,\dots,\psi_k\}} \rangle q_{\psi_i}) \\
\Delta(q_{[ret]R_i}, \sigma) &= \vee_{\phi', \psi_{1 \le j \le k}} ([ret, q_{\langle call \rangle \phi'\{\psi_1,\dots,\psi_k\}}] q_{\psi_i} \\
&\qquad \vee [ret, q_{[call]\phi'\{\psi_1,\dots,\psi_k\}}] q_{\psi_i})
\end{aligned}
$$

The priority of states of the form $q_{\mu X.\varphi}$ and $q_{\nu X.\varphi}$ are respectively odd and even, and roughly equal to the alternation depth of φ. The priority for all other states is 0. We do not define the acceptance condition in detail as we do not prove its correctness in this setting—this is because this part of the proof is exactly the same as in the translation from the μ-calculus to alternating parity tree automata.

We will, however, discuss in some more detail the transition relation of $\mathcal{A}_\varphi$. The automaton implements the operational semantics of the formula φ. If $\varphi = p$ or $\varphi = \neg p$, $\mathcal{A}_\varphi$ checks if the atomic proposition p holds at the node of the nested tree currently being read. Conjunction and disjunction in φ is captured respectively by conjunction and disjunction in the transition relation of $\mathcal{A}_\varphi$. If $\varphi = X$, then in the operational semantics of φ, the fixpoint formula binding X is executed recursively. Accordingly, $\mathcal{A}_\varphi$ loops to the state $\Psi(X)$. At a fixpoint formula of the form $\mu X.\varphi'$ or $\mu X.\varphi'$, the automaton starts the corresponding fixpoint computations by moving to the state for φ'. All this is exactly the same as for the μ-calculus and alternating tree automata. The difference from the tree setting is in the treatment of modal operators.

If $\varphi = \langle loc \rangle \varphi'$, the corresponding automaton makes a transition to the state for φ' along some *loc*-edge from the current node. The $[loc]$ modality is similar. Now suppose $\varphi = \langle call \rangle \varphi' \{\psi_1, \dots, \psi_k\}$. In this case $\mathcal{A}_\varphi$ transitions to the state for the formula φ' along some *call*-edge from the current node (the $[call]$ modality is similar, except the automaton sends copies along all *call*-edges). The constraint is that the automaton must be at the state q_{ψ_i} at the jump-successors of the current node marked by color i. The automaton checks this constraint using its $\langle ret \rangle$ and $[ret]$ modalities. Consider a formula $\langle ret \rangle R_i$ that asserts that some *ret*-successor of the current node has color i. Consider such a successor t and its jump-predecessor s. The automaton checks, using a disjunction, if the automaton state at node s (in the current copy of the automaton) corresponds to *any* subformula of φ starting with a $\langle call \rangle$ or $[call]$ modality. If so—i.e., if a formula of the form $\langle call \rangle \varphi' \{\psi_1, \dots, \psi_k\}$ or

$[call]\varphi'\{\psi_1, \ldots, \psi_k\}$ was asserted at the jump-predecessor s of this *ret*-successor—the automaton has to check that the nested tree from node t on satisfies the i-th return obligation—i.e., ψ_i—asserted by that formula. Accordingly, it changes state to q_{ψ_i}. $\square$

Theorem 13. *Given any AP-NTA $\mathcal{A}$, one can construct an* NT-μ *formula $\varphi_{\mathcal{A}}$ such that for any nested tree $\mathcal{T}$, $\mathcal{T} \in \mathcal{L}(\varphi_{\mathcal{A}})$ iff $\mathcal{T} \in \mathcal{L}(\mathcal{A})$. The size of $\varphi_{\mathcal{A}}$ is polynomial in the size of φ.*

We will first establish the above for alternating nested tree automata $\mathcal{A}$ accepting by a *final state* q_f. The reason is that the proof is simpler and more intuitive in this case. After this we will present the full proof.

Proof. It will be simpler for us to write the formula $\varphi_{\mathcal{A}}$ using a set of equations rather than in the standard form. Translation from this equational form to the standard form is as for the modal μ-calculus [GTW02].

Let $Q = \{q_1, \ldots, q_n\}$ and TT respectively be the sets of states and transition conditions of $\mathcal{A}$. For each $q \in Q$, we have a marker R_q; for each pair of states $q, q' \in Q$, we have a variable $X_{q,q'}$. Intuitively, a summary $\langle s, U_{q_1}, \ldots, U_{q_n}\rangle$ is collected in $X_{q,q'}$ iff $\mathcal{A}$ has a way to start at node s at state q, and end up at a return $s' \in U_{q_j}$ in state q_j, having checked that q' was the state of the automaton in the current play at the jump-predecessor of s'. Now for each pair of states $q, q' \in Q$, we define a map $\mathcal{F}_{q,q'} : TT \to \Phi$, where Φ is the set of NT-μ formulas:

$$\begin{aligned}
\mathcal{F}_{q,q'}(tt) &= tt \\
\mathcal{F}_{q,q'}(ff) &= ff \\
\mathcal{F}_{q,q'}(f_1 \wedge f_2) &= \mathcal{F}_{q,q'}(f_1) \wedge \mathcal{F}_{q,q'}(f_2) \\
\mathcal{F}_{q,q'}(f_1 \vee f_2) &= \mathcal{F}_{q,q'}(f_1) \vee \mathcal{F}_{q,q'}(f_2) \\
\mathcal{F}_{q,q'}(\langle call\rangle q'') &= \langle call\rangle (X_{q'',q})\{X_{q_1,q'}, \ldots, X_{q_n,q'}\} \\
\mathcal{F}_{q,q'}([call] q'') &= [call](X_{q'',q})\{X_{q_1,q'}, \ldots, X_{q_n,q'}\}
\end{aligned}$$

$$\begin{array}{rcl}
\mathcal{F}_{q,q'}(\langle loc\rangle q'') & = & \langle loc\rangle X_{q'',q'} \\
\mathcal{F}_{q,q'}([loc]q'') & = & [loc]X_{q'',q'} \\
\mathcal{F}_{q,q'}(\langle ret, q\rangle q'') & = & \langle ret\rangle(R_{q''}) \\
\mathcal{F}_{q,q'}([ret, q]q'') & = & [ret]R_{q''}
\end{array}$$

Then the formula $\varphi_{\mathcal{A}}$ is the formula corresponding to X_{q_0,γ_0} when taking the least fixpoint of the following equations:

$$X_{q,q'} = \begin{cases} tt & \text{if } q = q_f \\ \bigvee_{\sigma\subseteq AP}((\wedge_{p\in\sigma}p) \wedge (\wedge_{p\notin\sigma}\neg p) \wedge \mathcal{F}_{q',q}(\Delta(q,\sigma) \vee \Delta_r(q,\sigma))) & \text{otherwise.} \end{cases}$$

□

Now we give the general proof. Our translation from AP-NTAs to NT-μ uses finite-state alternating parity tree automata, which we will define now. The automata we use are bisimulation-closed and run on unranked trees—similar definitions may be found in Kirsten's survey chapter [GTW02] on alternating tree automata. For a set I of tags, a set Q of states, and a set AP of atomic propositions, let a *(finite-state) transition term* be of the form $g = p \mid \neg p \mid q \mid \langle a\rangle g \mid [a]g \mid g \wedge g \mid g \vee g$, where $a \in I$, $p \in AP$, and $q \in Q$. Fix a set TT^{at} of such terms. A finite-state alternating parity tree automaton (TA) over TT^{at} is a structure $\mathcal{M} = (Q, q_0, \gamma_0, \Delta, \Omega)$, where Q is a finite set of *automaton states*, $q_0 \in Q$ is the initial state, $\Delta : Q \to TT^{at}$ is a *transition function*, and $\Omega : Q \to \{1, 2, \ldots, n\}$ is the *parity accepting condition* that assigns a *priority* to each automaton state.

TAs run on unordered infinite trees whose nodes are labeled by 2^{AP} and edges by I. We skip a formal definition of the semantics. Intuitively, the term p means that the proposition p holds at the current node, $\langle a\rangle q$ means that the automaton propagates the state q along some a-labeled edge, and the term $[a]q$ means that the state q is passed along all a-edges. Note that we allow complex terms like $[a_1]\langle a_2\rangle q$, which means that the automaton first takes all edges labeled a_1, and then, from each child of the current node so reached, picks an edge labeled a_2 and passes the state q

to it. Terms can be combined conjunctively and disjunctively, as is par course for alternating automata. Our acceptance condition is max-parity, meaning a run is accepting if along all its infinite paths, the maximum priority seen infinitely often is even. We write $\mathcal{T} \models \mathcal{M}$ if a labeled tree $\mathcal{T}$ is accepted by $\mathcal{M}$.

Our proof also depends on a translation from TAs to the modal μ-calculus (actually, we will be interested in a syntactic fragment of the μ-calculus). Recall that formulas of the μ-calculus over a set of variables Var have syntax

$$\varphi = p \mid \neg p \mid \langle a \rangle \varphi \mid [a]\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid X \mid \mu X.\varphi \mid \nu X.\varphi,$$

where $a \in I$, $X \in Var$ and $p \in AP$.

Such formulas are interpreted over $(2^{AP}, I)$-labeled trees under environments $\mathcal{E} : X \mapsto S'$, where S' is a set of nodes in the tree in question. Again, we skip a formal definition of the semantics. We write $T, \mathcal{E} \models \varphi$ if a tree $\mathcal{T}$ satisfies the formula φ under an environment $\mathcal{E}$ (sometimes we write $t, \mathcal{E} \models \varphi$ if $\mathcal{T}$ in the above is rooted at the node t). If φ is closed, we can omit $\mathcal{E}$.

Now we define a special class of labeled trees obtained by applying a "summarization" transformation to nested trees. Speaking roughly, these trees have summaries as their nodes (in addition to some intermediate nodes), and trees rooted at returns are "plucked" and "lifted" to the matching call-sites. Formally, let NT be the set of all nested trees and $\mathbb{T}$ the set of all node and edge-labeled trees. We define a map $Summarize : NT \times \mathbb{N} \to \mathbb{T}$.

Let $k \geq 0$ and $I = \{call, ret, loc\}$. We consider the augmented set of atomic propositions $\widehat{AP} = AP \cup \{leaf_i : 1 \leq i \leq k\}$, and the set of edge labels $\widehat{I} = I \cup \{choose, in\} \cup \{i : 1 \leq i \leq k\}$. Then for a nested tree $\mathcal{S} = (((S, s_0, \rightarrow), \hookrightarrow), \lambda)$ whose nodes are labeled by the alphabet 2^{AP}, we define a node and edge-labeled tree $\mathcal{T} = Summarize(\mathcal{S}, k)$. Let $Summ_k$ be the set of k-colored summaries in $\mathcal{S}$. Then the set of nodes of $\mathcal{T}$ is $T \subseteq (Summ_k^{\mathcal{S}} \cup T')^+$, where $T' = \{(U_1, U_2, \ldots, U_k) :$ for all $1 \leq j \leq k$, $U_j \subseteq ME(u)$ for some $u \in S\}$ (the node set will be defined more precisely soon). The root of $\mathcal{T}$ is $t_0 = \langle s_0, \emptyset, \ldots, \emptyset \rangle$. The edges of the tree are labeled by the alphabet $\widehat{I}$ using a map $\eta_{\mathcal{T}}$; the nodes are labeled by the alphabet $\widehat{\Sigma} = 2^{\widehat{AP}}$ using a map $\lambda_{\mathcal{T}}$.

The set T of nodes and the set $E_{\mathcal{T}}$ of edges of $\mathcal{T}$ are those obtained when the computation described below reached a fixpoint. The set *Leaves* of leaf-nodes where we chop $\mathcal{S}$ is also obtained as part of the same least fixpoint. Initially, $T = \{t_0\}$ and $\mathit{Leaves} = E_{\mathcal{T}} = \emptyset$. Now, let us write $t \overset{a}{\rightarrowtail} t'$ if there is an a-labeled edge from node t to node t' in $\mathcal{T}$. Then:

1. For each s' such that $s \xrightarrow{loc} s'$ in $\mathcal{S}$ and each node $t.\mathbf{s} \notin \mathit{Leaves}$ in $\mathcal{T}$, where $\mathbf{s} = \langle s, U_1, U_2, \ldots, U_k\rangle$ is a summary in $\mathcal{S}$, we add a node $t.\mathbf{s}.\mathbf{s}'$ to T, where $\mathbf{s}' = \langle s', U_1', U_2', \ldots, U_k'\rangle$ such that $U_j' = U_j \cap MR(s')$ for all j. Also, we add an edge $t.\mathbf{s} \overset{loc}{\rightarrowtail} t.\mathbf{s}.\mathbf{s}'$ to $\mathcal{T}$.

2. For each s' such that $s \xrightarrow{call} s'$ in $\mathcal{S}$ and each node $t.\mathbf{s} \notin \mathit{Leaves}$, where $\mathbf{s} = \langle s, U_1, U_2, \ldots, U_k\rangle \notin \mathit{Leaves}$ is a summary of $\mathcal{S}$:

   (a) we add to T the node $t_1 = t.\mathbf{s}.s'$. We also add the edge $t.\mathbf{s} \overset{choose}{\rightarrowtail} t_1$.

   (b) For every t_1 as above, we add to T every node that is of the form $t_2 = t_1.(V_1, V_2, \ldots, V_k)$, where $V_1, \ldots, V_k \subseteq \mathit{ME}(s')$. We also add edges $t_1 \overset{choose}{\rightarrowtail} t_2$ for each such t_2.

   (c) For every t_2 as above, we add to T the node $t_3 = t_2.\mathbf{s}'$, where $\mathbf{s}' = \langle s', V_1, V_2, \ldots, V_k\rangle$, and for every $s'' \in V_j$ for some j, we add the node $t_4 = t_2.\mathbf{s}''$, where $\mathbf{s}'' = \langle s'', U_1', U_2', \ldots, U_k'\rangle$ is a summary such that $U_i' = U_i \cap \mathit{ME}(s'')$. We also add the edge $t_2 \overset{in}{\rightarrowtail} t_3$, and, for each t_4 as above such that $s'' \in V_j$, the edge $t_2 \overset{j}{\rightarrowtail} t_4$.

   Nodes of the form $t.s$ and $t.(V_1, \ldots, V_k)$, where $s \in S$, $t \in \{\epsilon\} \cup T$ and $V_1, \ldots, V_k \subseteq T$, will be sometimes referred to as *intermediate nodes*.

3. For each s' such that $s \xrightarrow{ret} s'$ in $\mathcal{S}$ and each node $t.\mathbf{s} \notin \mathit{Leaves}$ of $\mathcal{T}$ such that $\mathbf{s} = \langle s, U_1, U_2, \ldots, U_k\rangle$ is a summary, we add the node $t.\mathbf{s}.\widetilde{s'}$ to T, where

$\widetilde{s'} = (s', U_1, U_2, \ldots, U_k)$. Note that $\widetilde{s'}$ is not a summary. We also add the edge $t.\mathbf{s} \overset{ret}{\rightarrowtail} t.\mathbf{s}.\widetilde{s'}$ to $E_{\mathcal{T}}$, and $t.\mathbf{s}.\widetilde{s'}$ to *Leaves*.

Note that the above also defines the edge-labeling function $\eta_{\mathcal{T}}$ in $\mathcal{T}$. The node-labeling map $\lambda_{\mathcal{T}} : T \to \widehat{\Sigma}$ is defined as: for $t \in T \setminus \mathit{Leaves}$, $\lambda_{\mathcal{T}}(t.\mathbf{s}) = \lambda(s)$ if $t \in \{\epsilon\} \cup T$ and $\mathbf{s} = \langle s, U_1, \ldots, U_k\rangle \in \mathit{Summ}_k^{\mathcal{S}}$, and $\lambda_{\mathcal{T}}(t.(V_1, \ldots, V_k)) = \lambda_{\mathcal{T}}(t.s) = \emptyset$ for all $t, s, V_1, \ldots, V_k$. For $t' = t.\widetilde{s} \in \mathit{Leaves}$, where $\widetilde{s} = (s, U_1, \ldots, U_k)$, we set $\lambda_{\mathcal{T}}(t') = \{\mathit{leaf}_j : s \in U_j\}$.

We will now proceed to the lemmas we will use in this proof. First, let AP be defined as before; now for a set Q of TA states, fix the set of atomic terms $TT^{at}(Q, AP)$, comprising the terms p, $\neg p$, $\langle loc\rangle q$, $[loc]q$, $\langle ret\rangle \mathit{leaf}_i$, $[ret]\mathit{leaf}_i$, $\langle call\rangle\langle choose\rangle([in]q \wedge [1]\ q_1 \ldots \wedge [k]\ q_k)$, and $[call]\langle choose\rangle([in]\ q \wedge [1]\ q_1 \ldots \wedge [k]\ q_k)$ for all $q, q_1, \ldots, q_k \in Q$, $1 \leq i \leq k$, and $p \in AP$. Then:

Lemma 4. *Given an AP-NTA $\mathcal{A}$, we can effectively construct a TA $\mathcal{F}(\mathcal{A})$ over $TT^{at}(Q, AP)$ such that for any nested tree $\mathcal{S}$, we have $\mathcal{S} \in \mathcal{L}(\mathcal{A})$ iff $\mathit{Summarize}(\mathcal{S}, k) \models \mathcal{F}(\mathcal{A})$. The size of $\mathcal{F}(\mathcal{A})$ is polynomial in the size of $\mathcal{A}$.*

The second lemma is an augmentation of the translation from TAs to the modal μ-calculus. Consider TAs over the set of atomic terms $TT^{at}(Q, AP)$ defined as above. Now consider the syntactic fragment of the μ-calculus, parameterized by TT^{at}, whose formulas φ are given by:

$$\begin{aligned}\varphi = p \mid \neg p \mid X \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \langle loc\rangle\ \varphi \mid [loc]\ \varphi \\ \mid \langle call\rangle\langle choose\rangle([in]\varphi \wedge [1]\varphi \wedge \ldots \wedge [k]\varphi) \\ \mid [call]\langle choose\rangle([in]\varphi \wedge [1]\varphi \wedge \ldots \wedge [k]\varphi) \\ \mid \langle ret\rangle \mathit{leaf}_i \mid [ret]\mathit{leaf}_i \mid \mu X.\varphi \mid \nu X.\varphi,\end{aligned}$$

where $p \in AP$ and $X \in \mathit{Var}$. Let $\Phi(TT^{at})$ be the set of formulas of the above form. Then:

Lemma 5. *For any TA $\mathcal{M}$ with state set Q and over the set of atomic terms $TT^{at}(Q, AP)$, we can construct a closed formula $\Psi(\mathcal{M}) \in \Phi(TT^{at})$ such that for any $(\widehat{\Sigma}, \widehat{I})$-labeled tree $\mathcal{T}$, we have $\mathcal{T} \models \Psi(\mathcal{M})$ iff $\mathcal{T} \models \mathcal{M}$. The size of $\Psi(\mathcal{M})$ is polynomial in the size of $\mathcal{M}$.*

The third lemma connects μ-calculus formulas of the above form to NT-μ formulas.

Lemma 6. *Let φ be a closed μ-calculus formula in $\Phi(TT^{at})$ defined above. Then from φ we can construct a* NT-μ *formula $Nest(\varphi)$ such that for any $(\widehat{\Sigma}, \widehat{I})$-labeled tree $\mathcal{T}$ satisfying $\mathcal{T} = Summarize(\mathcal{S}, k)$ for some nested tree $\mathcal{S}$ over 2^{AP}, we have $\mathcal{T} \models \varphi$ iff $\mathcal{S} \in \mathcal{L}(Nest(\varphi))$. The size of $Nest(\varphi)$ is polynomial in the size of φ.*

Let us first establish that if the above three lemmas hold, then Theorem 13 holds. This is proved by the following construction. Given $\mathcal{A}$, we construct the NT-μ formula $\varphi_{\mathcal{A}} = Nest(\Psi(\mathcal{F}(\mathcal{A})))$. By the above lemmas, this construction is possible. Now fix any structured tree $\mathcal{S}$. If $\mathcal{S} \models \mathcal{A}$, then by Lemma 4, $Summarize(\mathcal{S}, k) \models \mathcal{F}(\mathcal{A})$, and by Lemma 5, $Summarize(\mathcal{S}, k) \models \Psi(\mathcal{F}(\mathcal{A}))$. But then by Lemma 6, $\mathcal{S} \models \varphi_{\mathcal{A}}$. Similarly, if $\mathcal{S} \models \varphi_{\mathcal{A}}$, it is easily established that $\mathcal{S} \models \mathcal{A}$. $\square$

Let us now prove Lemmas 4-6.

Proof of Lemma 4. Let $\mathcal{A} = (Q, q_0, \Delta, \Omega : Q \to \{1, \ldots, n\})$ be an AP-NTA. Let $\perp$ be a special state not in Q. Then states of $\mathcal{F}(\mathcal{A})$ are of the form $r = (q, \gamma, \theta, m)$, where $q \in Q$, $\gamma \in Q \cup \{\perp\}$, and $theta, m \in \{1, \ldots, n\}$ are priorities of $\mathcal{A}$. Let us now consider a map $indexmap : Q \times \{1, \ldots, n\} \to \{1, \ldots, |Q|n\}$ that assigns a unique index to every tuple (q, θ). Intuitively, *indexmap* assigns an index to the fact that "$\mathcal{A}$ can land up at a matching exit of the current node in state q, with θ as the minimum priority seen in the current context."

We will present the semantics of some of the interesting transitions of $\mathcal{F}(\mathcal{A})$ in English. Since treatment of alternation is similar in AP-NTAs and TAs, we will focus on "atomic" transition terms. Suppose $\mathcal{F}(\mathcal{A})$ is in state $r = (q, \gamma, \theta, p)$ while reading a node t of $\mathcal{T}$. Let us now have, say, $\Delta(q, \lambda(t)) = \langle call \rangle q'$ or $\Delta(q, \lambda(t)) = [call]q'$, In these two cases, $\mathcal{F}(\mathcal{A})$ forks copies to check that the $\lambda(t)$ is precisely the set of propositions satisfied at t; it also forks a copy that respectively reads one or all of the *call*-children of t. Each of these children are intermediate nodes, each with (uncountably) many *choose*-children. $\mathcal{F}(\mathcal{A})$ now reads *one* of these *choose*-children; once this is done, it is at an intermediate node with outgoing transitions labeled $in, 1, \ldots, k$. At this point $\mathcal{F}(\mathcal{A})$ passes:

1. the state $(q', q, 0, \Omega(q'))$ to the unique child along the edge labeled *in*;
2. the state $(q'', q, \max(\theta, m), m)$ to *every* child along a transition labeled j, for every j, iff $j = indexmap(q'', m)$. Now the move is over.

If $\Delta(q, \lambda(t)) = \langle loc \rangle q'$ or $[loc]q'$, then from state r, respectively along some or all *loc*-edges, the TA $\mathcal{F}(\mathcal{A})$ passes the state $(q', z, \max(\theta, \Omega(q)), \Omega(q))$ (also, copies need to be forked to ensure that $\lambda(t)$ is satisfied currently).

Let us now assume that $\Delta(q, \lambda(t)) = \langle ret, q'' \rangle q'$ or $[ret, q'']q'$. Again, it is made sure that $\lambda(t)$ holds currently. Now, if r is the current state of $\mathcal{F}(\mathcal{A})$, the state $(q', q'', \max(\theta, \Omega(q)), \Omega(q))$ is passed respectively along some or all *ret*-edges out of t. By definition of $\mathcal{T}$, the child $t' = t.\widetilde{s'}$ is a leaf. At this point $\mathcal{F}(\mathcal{A})$ accepts if $leaf_j \in \lambda_{\mathcal{T}}(t')$, where $j = indexmap(q', \max(\theta, \Omega(q)))$, and rejects otherwise.

The priority function $\widehat{\Omega}$ of a state $r = (q, \gamma, \theta, m)$ is defined as: $\widehat{\Omega}(r) = m$.

The TA $\mathcal{F}(\mathcal{A})$ is clearly over the restricted set of transition terms $TT^{at}(Q, AP)$ that we fixed. Lemma 4 may now be established using arguments used by Walukiewicz in the context of pushdown parity games [Wal01]. Finally, note that $\mathcal{F}(\mathcal{A})$ is polynomial in the size of $\mathcal{A}$. $\square$

Proof of Lemma 5. We follow a translation from TAs to the μ-calculus discussed in a survey chapter by Alberucci [GTW02]. The proof proceeds by induction on the *index* n of the TA $\mathcal{M}$, defined as the maximum number of distinct priorities in a strongly connected component in the state transition graph of $\mathcal{M}$. It also uses a lemma about the existence of simultaneous fixpoints. We will not reproduce the arguments here in detail, but will present a skeleton.

First, we note that the simultaneous fixpoint theorem holds even for the subset of the μ-calculus we consider here. More precisely:

Lemma 7. *Let $T_1, \ldots, T_k$ be sets of nodes of $\mathcal{T}$, and let $\delta_1, \ldots, \delta_k \in \Phi(TT^{at})$ be μ-calculus formulas in $\Phi(TT^{at})$ with k free variables each. Now consider the monotone map $F : (T_1, \ldots, T_k) \mapsto ([\![\delta_1(T_1, \ldots, T_k)]\!], \ldots, [\![\delta_k(T_1, \ldots, T_k)]\!])$. There are μ-formulas $\tau_1, \ldots, \tau_k \in \Phi(TT^{at})$ and ν-formulas $\rho_1, \ldots, \rho_k \in \Phi(TT^{at})$ such that $(\tau_1, \ldots, \tau_k)$ and $(\rho_1, \ldots, \rho_k)$ are respectively the least and greatest fixpoints of F.*

The proof of this follows from the fact that simultaneous fixpoints are obtained by substitution of variables by formulas, so that the basic structure of the modalities remains unaltered.

Let us now go through the induction. The idea is to inductively replace states appearing in infinite cycles by variables. Base case: $n = 0$, so that the automaton does not have any cycles in its state transition graph. In this case the formula $\Psi(\mathcal{M})$ is obtained by "expanding" the transitions; given the structure of TT^{at}, we conclude that the $\Psi(\mathcal{M})$ is in $\Phi(TT^{at})$.

For the induction step, consider the TA $\mathcal{M}(Q', q')$ obtained by duplicating the state $q' \in Q' \subseteq Q$ and declaring q' as the new initial state, and then replacing all states $q \in Q'$ by propositions. Intuitively, this operation identifies the runs of $\mathcal{M}$ starting from state q', while "chopping off" said runs at states in Q'. Clearly, $\mathcal{M}(Q', q')$ is also a TA over TT^{at}; it turns out that $\mathcal{M}(Q', q')$ has index less than n and hence can be translated to an equivalent formula in $\Phi(TT^{at})$. Now we let Q' be the set of states of maximum priority in $\mathcal{M}$, set q_i to be the i-th state in Q', and consider the formula $\delta_i \in \Phi(TT^{at})$ equivalent to $\mathcal{M}(Q', q_i)$. We do this for all i, then plug these formulas δ_i into Lemma 7 and obtain the formulas $\tau_1, \ldots, \tau_k$ corresponding to their simultaneous fixpoints (depending on whether the maximum priority is odd or even, we will need least or greatest fixpoints). We also look at the formula τ_0 equivalent to $\mathcal{M}(Q')$, which is simply $\mathcal{M}$ with states in Q' moved to propositions.

Then $\Psi(\mathcal{M}) = \tau_0(\tau_1, \ldots, \tau_k)$, which, by our arguments, belongs to $\Phi(TT^{at})$. Further, as the proof in Alberucci's chapter shows, $\Psi(\mathcal{M})$ is polynomial in the size of $\mathcal{M}$. $\square$

Proof of Lemma 6. For subformulas of a μ-calculus formula φ in $\Phi(TT^{at})$, let us define $\mathit{Nest}(\varphi)$ inductively:

$$
\begin{array}{lrcl}
(1) & \mathit{Nest}(p) & = & p \\
(2) & \mathit{Nest}(\neg p) & = & \neg p \\
(3) & \mathit{Nest}(X) & = & X \\
(4) & \mathit{Nest}(\varphi_1 \wedge \varphi_2) & = & \mathit{Nest}(\varphi_1) \wedge \mathit{Nest}(\varphi_2) \\
(5) & \mathit{Nest}(\varphi_1 \vee \varphi_2) & = & \mathit{Nest}(\varphi_1) \vee \mathit{Nest}(\varphi_2) \\
(6) & \mathit{Nest}(\langle \mathit{loc} \rangle \varphi) & = & \langle \mathit{loc} \rangle \mathit{Nest}(\varphi) \\
(7) & \mathit{Nest}([\mathit{loc}]\ \varphi) & = & [\mathit{loc}]\ \mathit{Nest}(\varphi) \\
(8) & \mathit{Nest}(\langle \mathit{call} \rangle \langle \mathit{choose} \rangle ([\mathit{in}]\varphi \wedge [1]\psi_1 \wedge \ldots \wedge [k]\psi_k)) & = & \langle \mathit{call} \rangle (\mathit{Nest}(\varphi)) \\
 & & & \{\mathit{Nest}(\psi_1), \ldots, \mathit{Nest}(\psi_k)\} \\
(9) & \mathit{Nest}([\mathit{call}] \langle \mathit{choose} \rangle ([\mathit{in}]\varphi \wedge [1]\varphi \wedge \ldots \wedge [k]\varphi)) & = & [\mathit{call}](\mathit{Nest}(\varphi)) \\
 & & & \{\mathit{Nest}(\psi_1), \ldots, \mathit{Nest}(\psi_k)\} \\
(10) & \mathit{Nest}(\langle \mathit{ret} \rangle \mathit{leaf}_i) & = & \langle \mathit{ret} \rangle R_i \\
(11) & \mathit{Nest}([\mathit{ret}]\ \mathit{leaf}_i) & = & [\mathit{ret}] R_i \\
(12) & \mathit{Nest}(\mu X.\varphi) & = & \mu X.\mathit{Nest}(\varphi) \\
(13) & \mathit{Nest}(\nu X.\varphi) & = & \nu X.\mathit{Nest}(\varphi)
\end{array}
$$

Now, let $\mathcal{S}$ be any nested tree and $\mathcal{T} = \mathit{Summarize}(\mathcal{S}, k)$ as before; recall that $\mathit{Summ}_k^{\mathcal{S}}$ is the set of summaries in $\mathcal{S}$ and T the set of nodes of $\mathcal{T}$. Consider a μ-calculus environment $\widehat{\mathcal{E}} : \mathit{Var} \to T$ for $\mathcal{T}$; for any two bisimilar nodes t_1, t_2 in

$\mathcal{T}$, we can assume that $t_1 \in \widehat{\mathcal{E}}$ iff $t_2 \in \widehat{\mathcal{E}}$. Now note that even if we prevent our environments from containing nodes not of the form $t.\mathbf{s}$ (i.e., if we remove every "non-summary" node from each of the fixpoint sets as they are computed), the semantics of φ at "summary" nodes of form $t.\mathbf{s}$ (and hence at the root of $\mathcal{T}$) is not affected. This can be verified by a simple induction. We will assume environments for $\mathcal{T}$ to have this property from now on. Now consider the NT-μ environment $\mathcal{E} : \mathit{Var} \to \mathit{Summ}_k^{\mathcal{S}}$ defined as: for all $X \in \mathit{Var}$, $\mathcal{E}(X) = \{\mathbf{s} : \mathbf{s}$ is a summary, and $t.\mathbf{s} \in \widehat{\mathcal{E}}(X)$ for some $t\}$.

Our inductive hypothesis is: for any environment $\widehat{\mathcal{E}}$ in $\mathcal{T}$, any node $t.\mathbf{s}$ in $\mathcal{T}$ where $\mathbf{s}$ is a summary, and any $\varphi \in \Phi(TT^{at})$, $\mathbf{s} \in [\![\mathit{Nest}(\varphi)]\!]_{\mathcal{E}}$ iff $t.\mathbf{s}, \widehat{\mathcal{E}} \models \varphi$.

Cases (1), (2), (4), (5), (6), (7), (10), and (11) are easy. For the case $\varphi = X$ (case 3), recall that $\widehat{\mathcal{E}}$ is bisimulation-closed. For cases (8) and (9) (the call-clauses), note that the branching of $\mathcal{T}$ exactly captures the semantics of the call clauses of NT-μ. For the case $\varphi = \mu X.\varphi'$ (the case $\nu X.\varphi'$ is similar), assume that T_i and BS_i are the i-th sets in the fixpoint computation in φ and $\mathit{Nest}(\varphi)$ (we can assume that T_i is bisimulation-closed). We show that $BS_i = \{\mathbf{s} : t.\mathbf{s} \in T_i\}$ by induction. Initially, $T_0 = \emptyset$ and $BS_0 = \emptyset$. Now, $T_{i+1} = \{t : t, \widehat{\mathcal{E}}[X := T_i] \models \varphi'\}$, and $BS_{i+1} = [\![\mathit{Nest}(\varphi')]\!]_{\mathcal{E}[X:=BS_i]}$. It is easy to see that $BS_{i+1} = \{\mathbf{s} : t.\mathbf{s} \in T_{i+1}\}$. It is also easily seen that $\mathit{Nest}(\varphi)$ is linear in the size of φ. This establishes our result. $\square$

4.2.1 Expressiveness of NT-μ: other results

Theorems 12 and 13 imply a few results for NT-μ that we have not already derived. Let us define the *satisfiability problem* for NT-μ as the problem of determining, given an NT-μ formula φ, if $\mathcal{L}(\varphi) = \emptyset$. Then we have:

Theorem 14. *Given a* NT-μ *formula, the problem of checking whether there is some nested tree that satisfies it is undecidable.*

Proof. By Theorem 11, determining if $\mathcal{L}(\mathcal{A}) = \emptyset$ is undecidable. However, by Theorem 13, for every AP-NTA $\mathcal{A}$ there is an NT-μ formula $\varphi_{\mathcal{A}}$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\varphi_{\mathcal{A}})$.

Thus, if we could determine, given an NT-μ formula φ, if $\mathcal{L}(\varphi) =$, then we could solve the emptiness problem for AP-NTAs. This means the satisfiability problem for NT-μ is undecidable. $\square$

The automata-theoretic characterization also makes it easy for us to compare the expressiveness of NT-μ with that of other temporal logics. As AP-NTAs are more expressive than alternating parity tree automata, which are polynomially interconvertible with closed μ-calculus formulas, we have:

Theorem 15. *Any closed μ-calculus formula φ may be converted into an equivalent* NT-μ *formula φ'. The size of φ' is polynomial in the size of φ.*

We may now also relate NT-μ to the temporal logic CARET [AEM04], which can express many linear-time context-sensitive properties of programs. This is because any CARET formula may be translated to an exponentially larger nondeterministic finite automaton on nested words. As such automata form a subclass of AP-NTAs, we have:

Theorem 16. *Any* CARET *formula φ may be translated to an equivalent* NT-μ *formula φ'. The size of φ' is at worst exponential in the size of φ.*

4.3 Monadic second-order logic on nested trees

We now study *monadic second-order (MSO) logic* interpreted on ordered nested trees. Formulas in MSO-logic are built over a set of first-order variables $(x, y \ldots)$ and a set of second-order variables $(X, Y, \ldots)$, ranging over nodes and sets of nodes in a nested tree $\mathcal{T}$. For each $\sigma \in \Sigma$, the signature of MSO logic has a unary predicate Q_σ, where $Q_\sigma(s)$ is true at a node s iff s is labeled by σ; we also have a binary equality predicate $x = y$. There are also left and right edge predicates $x \rightarrow_1 y$ and $x \rightarrow_2 y$, and a jump-edge predicate $x \hookrightarrow y$.

The syntax of MSO logic is: $\varphi := Q_\sigma(x) \mid \neg\varphi \mid \varphi \vee \varphi \mid x = y \mid x \rightarrow_1 y \mid x \rightarrow_2 y \mid x \hookrightarrow y \mid \exists x.\varphi \mid \exists X.\varphi \mid X(x)$. The semantics is the natural one on ordered nested trees. The language $\mathcal{L}(\varphi)$ of φ is the set of nested trees that satisfy it; φ is said to be bisimulation-closed if $\mathcal{L}(\varphi)$ is bisimulation-closed. The model checking problem is: given φ and a nested state machine $\mathcal{M}$, does the unfolding $\mathcal{T}_{ord}(\mathcal{M})$—defined as an ordered nested tree— satisfy φ?

While MSO logic over trees is decidable [GTW02], MSO logic over nested tree structures is not:

Theorem 17. *The model checking problem for (even the bisimulation-closed fragment of) MSO-logic on nested state machines is undecidable.*

Proof. We note that the gadget $\mathcal{S}$ used to prove Theorem 11 (Figure 4.2-(b)) may be embedded in the ordered unfolding $\mathcal{T}$ of a simple nested state machine. Using existential set quantification, MSO-logic can select $\mathcal{S}$ from $\mathcal{T}$, so that there is an MSO-logic formula φ that holds on $\mathcal{T}$ iff the substructure $\mathcal{S}$ exists. We note that the only extra expressive power we need over an AP-NTA is that of existential quantification, so that φ is bisimulation-closed. The theorem follows. $\square$

The satisfiability problem for MSO-logic on nested trees is also undecidable.

Now we compare the expressiveness of MSO-logic on nested trees with that of nested tree automata. We can show that:

Theorem 18. *There is a bisimulation-closed MSO-logic formula φ such that there is no AP-NTA $\mathcal{A}$ satisfying $\mathcal{L}(\mathcal{A}) = Unord(\mathcal{L}(\varphi))$.*

Proof. Consider the problem of checking if a fixed Turing machine TM halts on an input word w, for some Turing machine for which this problem is undecidable. It is possible to build a simple NSM $\mathcal{M}$ and a bisimulation-closed MSO-logic formula φ such that $\mathcal{T}(\mathcal{M})$ satisfies φ iff TM has a halting run on w. Now, if this MSO-logic formula were expressible by an AP-NTA $\mathcal{A}$, then we could decide whether TM halts on w by model checking $\mathcal{M}$ against $\mathcal{A}$. This is of course impossible. The theorem follows as a result. $\square$

Note now that the MSO formula φ above recognizes a language that is a *projection* of an AP-NTA language. This is because aside from the existential quantifier with which φ selects a substructure from the unfolding of an NSM, there is nothing in φ that cannot be captured by an AP-NTA. However, for an MSO formula ψ, the language $\mathcal{L}(\exists X.\psi)$, for a second-order variable X, is the *projection* of the language $\mathcal{L}(\psi)$. Thus, Theorem 18 establishes that AP-NTAs are not closed under projection (Theorem 9).

We have shown that MSO-logic on nested trees can express languages that AP-NTAs (and therefore NP-NTAs) cannot. Now we show that MSO-logic is strictly more expressive than NP-NTA:

Theorem 19. *For every NP-NTA $\mathcal{A}$, there is an MSO-logic formula $\varphi_{\mathcal{A}}$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\varphi)$.*

Proof. We encode runs of any given NP-NTA $\mathcal{A}$ by an MSO-logic formula $\varphi_{\mathcal{A}}$. The latter uses existential quantification over sets to "guess" a global labeling of the nodes of a nested tree by states of $\mathcal{A}$. The predicates $\longrightarrow$ and $\hookrightarrow$ are used to check the consistency of this guess. $\square$

However, a "jump-edge" predicate seems too weak to capture the interplay of recursion and Boolean closure in AP-NTAs; higher-order quantification seems necessary. We conjecture that there is a language L recognized by an AP-NTA such that there is no MSO formula φ that recognizes L_{ord}, where $Unord(L_{ord}) = L$, making MSO neither less nor more expressive than AP-NTAs.

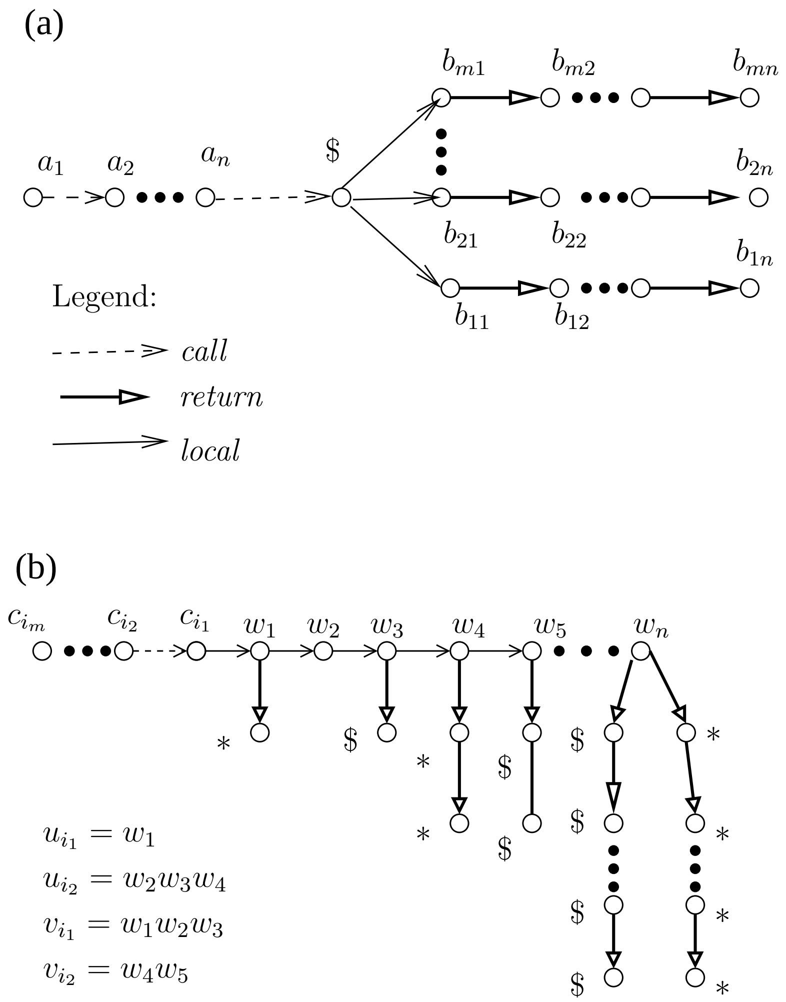

Figure 4.2: (a) Expressiveness of AP-NTAs and NP-NTAs (b) Gadget for undecidability

Chapter 5

The Pal specification language

This chapter presents a specification language—called Pal—and a tool to instrument C code using it. Monitors in Pal are based on nested word automata and can express context-sensitive safety requirements. The language extends the Blast specification language [BCH$^+$04], and while its richer foundations lets it state context-sensitive properties, it has syntax close to Blast's and allows easy instrumentation. Once a program has been instrumented with a Pal monitor, it can be used for testing or run-time verification as well as static analysis or model checking.

The chapter is organized as follows. In Section 5.1, we introduce context-sensitive safety properties, and in Section 5.2, we describe the Pal language. Next, in Section 5.4, we discuss its implementation and present a few case studies.

5.1 Context-sensitive safety properties

A safety property is a program requirement that asserts that something bad never happens during any execution of the program. A more formal definition is: a safety property is a requirement φ such that if φ is violated, then there is a finite counterexample in which the violation occurs. An example requirement is: "No program state satisfying a proposition p is ever reached." If such a state is indeed reached at some point, the finite execution up to that point serves as a counterexample to the property.

The class of temporal safety properties can be captured automata-theoretically. Let L_c be the language of counterexamples to the requirement—by definition, this is a language of finite executions. In traditional formal methods, where executions are modeled by words and requirements are based on regular word or tree languages, L_c is a regular language of finite words. If L is the language of executions of the abstraction, then the verification problem involves determining if $L \cap L_c = \emptyset$.

Recall that in classical model checking, programs are abstracted by finite-state machines. This makes L, and by implication $(L \cap L_c)$, a regular language (of finite words). Checking its emptiness then amounts to determining the reachability of a final state in its automaton representation. This view continues in context-sensitive model checkers like Blast [HJM+02] and Slam [BR01]. In this case the programmer specifies a program using a word automaton, or *monitor*, that accepts all unsafe program behaviors. Letting L be the language of program executions, the problem is once again to check if $L \cap L_c$— the language of the automaton that is the "product" of the program and the specification— is empty. While this captures only a restricted set of program requirements, this is an important subclass. In fact, the most successful software model checkers so far—Slam and Blast— restrict themselves to the model checking of safety properties.

However, as we have argued in this thesis, regular word languages are insufficient for many interesting properties of procedural programs. For example, while a traditional monitor can state a property about a global path in a program, it cannot express requirements on *local paths.* While we can extend the power of specifications by allowing ω-regular requirements, we can also extend them by allowing context-sensitive specifications. In fact, these two axes are orthogonal: we can, for example, think of context-sensitive liveness properties, context-sensitive fairness properties, etc. Accordingly, we have built a specification language that can state *context-sensitive* *safety requirements*—properties whose counterexamples are given by finite nested words. Lifting the definition from classical model checking, we define a context-sensitive safety property to be one whose negation is given by a *regular language of finite nested words* [AM04, AM06]. Typical properties include:

- Something bad never happens along a *local program path*. Recall the property $AG^l_c p$ from Chapter 3. This is an example of a context-sensitive safety property.

- It is impossible to reach a program state where the stack satisfies a certain regular requirement. For example, consider the stack inspection property in Chapter 3. This is a context-sensitive safety property as well.

Nested word monitors

Recall the definition of nested words from Chapter 2. A Pal monitor operates on nested words. Formally, a monitor on nested words labeled by an input alphabet Σ is defined as a tuple $\mathcal{A} = \langle Q, \Sigma, q_{in}, \delta_{loc}, \delta_{call}, \delta_{ret}, F \rangle$, where Q is a (possibly infinite) set of states Q, q_{in} is the initial state, and $\delta_{loc} \subseteq Q \times \Sigma \times Q$, $\delta_{call} \subseteq Q \times \Sigma \times Q$, and $\delta_{ret} \subseteq Q \times Q \times \Sigma \times Q$ are the local, call and return transition relations. The *acceptance condition* $F \subseteq Q$ is a set of *final states*. If $(q, \sigma, q') \in \delta_{loc}$ for some $q, q' \in Q$ and $\sigma \in \Sigma$, then we write $q \xrightarrow{loc,\sigma} q'$; if $(q, \sigma, q') \in \delta_{call}$, we write $q \xrightarrow{call,\sigma} q'$; if $(q, q', \sigma, q'') \in \delta_{ret}$, we write $(q, q') \xrightarrow{ret,\sigma} q''$. We require that the transition relations are deterministic— e.g., for a fixed state q and input symbol σ, there is only one transition of the form $q \xrightarrow{loc,\sigma} q'$.
The monitor $\mathcal{A}$ starts in the initial state, and reads a nested word from left to right. At a call or local position, the current state is determined by the state and the input symbol at the previous position, while at a return position, the current state can additionally depend on the state of the run just before processing the symbol at the jump-predecessor. Formally, the *run* ρ of the monitor $\mathcal{A}$ over a finite nested word $\mathcal{W} = (\sigma_1\sigma_2 \ldots, \hookrightarrow)$ is the finite sequence $q_0, q_1, q_2, \ldots q_k$ over Q such that $q_0 = q_{in}$, and:

- for all $0 \leq i < k$, if i is a call position of $\mathcal{W}$, then $(q_i, \sigma_i, q_{i+1}) \in \delta_{call}$;
- for all $0 \leq i < k$, if i is a local position, then $(q_i, \sigma_i, q_{i+1}) \in \delta_{loc}$;
- for $2 \leq i \leq k$, if i is a return position with jump-predecessor j, then $(q_{i-1}, q_{j-1}, \sigma_i, q_i) \in \delta_{ret}$.

Note that due to determinism, the monitor has a single run over a given nested word. The nested word $\mathcal{W}$ is *accepted* by $\mathcal{A}$ if the last state q_k in this run belongs to F (recall, however that if a monitor accepts a nested word modeling an execution iff the execution *violates* a safety property).

We note that if $\mathcal{A}$ has only finitely many states, then it is a deterministic finite automaton over finite nested words [AM04, AM06]. In this case it accepts a *regular language of nested words.*

5.2 Instrumenting C code

Let us now shift our focus to the problem of analyzing C programs. A popular approach taken by monitor languages in this setting is to express the monitor in C syntax. The states of a monitor are encoded by a set of C variables, which can be modified by C code capturing the transitions. For example, consider the safety requirement: "No two calls to an action *write()* happen in an execution without a call to *read()* in between." This requirement is captured by the word automaton in Figure 5.1, which has three states (one of which is an error state), and has input symbols *read* and *write* corresponding to the observables of the monitored program. The state of this automaton may be encoded by a C variable *state*; the action taken by the automaton on reading an input symbol can be encoded by a block of C code that tests the current value of this variable and modifies it (note that for this to be possible, the monitor has to be deterministic). For example, the code for the automaton's transition on a *write* input symbol is given in the same figure.

```
int state;

write:

    if (state == 0)
        state = 1;
    else if (state == 1)
        error();
```

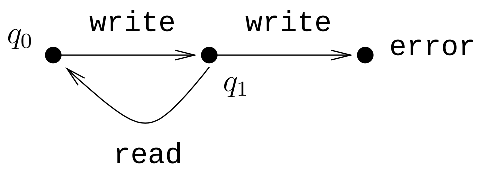

Figure 5.1: A safety monitor

Now consider the "product" of this monitor and a C program. An execution of this product comprises executions of the monitor and the program in lock-step, with the monitor firing transitions on certain program actions. Now note that a program behaving in this manner can be obtained by *instrumenting* the input program with monitor code. For instance, to instrument a program with the monitor in Figure 5.1, we add the single global variable *state* to the program— this variable has value 0 or 1 depending on whether the automaton's state is q_0 or q_1. Also, we inject code for monitor transitions immediately before every *read()* or *write()* statement in the program. Note that the added monitor variable has to be global, as it may be modified at any point of the execution.

We observe that this instrumented program fails an assertion if and only if the input program violates the safety specification. Also note that there are many static and dynamic, formal and informal techniques to detect assertion failures— including abstraction-based model checking as well as testing or run-time verification. We may use any one of these techniques to determine if program satisfies its requirement.

Let us now consider a context-sensitive safety requirement: "No call to *write()* happens while the routine *foo* is in the stack." We note that this property may be expressed by a nested word monitor with states q_0, q_1 and *error*. We assume that every entry and exit point of the procedure *foo* is labeled with an input symbol *foo*.

```
global int state = 0;

{
   local int stored;
   stored = state;
}
   foo ();
{
   state = stored;
}
```

Figure 5.2: Instrumenting with nested word automata

Then the transitions are:

$$q_0 \xrightarrow{call,foo} q_1$$
$$(q, q_0) \xrightarrow{ret,foo} q_0$$
$$(q, q_1) \xrightarrow{ret,foo} q_1$$
$$q_1 \xrightarrow{loc,write} error,$$

where $q \in \{q_0, q_1\}$.

We will now show a way to implement this monitor so that it can be used for easy instrumentation of C code. Let us now instrument a statement in the input program by injecting blocks of code *before* and *after* it. These two blocks serve as pre and post-conditions for the statement in the product program. Further, we allow these two blocks two share *local variables*. In an execution of the instrumented program, the monitor's state, represented as a global variable, is stored in a local variable before the call to *foo* (see Figure 5.2). This value can be retrieved at the matching return of this call— and used in its postcondition. In general under this scheme, a monitor may update its state along the global program execution via assignments to its global variables, as well as retrieve its state at the matching call when transitioning to a return. Thus, a monitor such as this encodes a nested word monitor.

5.3 The PAL language

We will now present the PAL language using a more involved example. Consider a C program that opens and closes files via calls to `fopen` and `fclose`, and the following requirement: "a secret file `dat` is not opened outside the scope of a file-handling routine `foo`. If `foo`, or a procedure called transitively from it, opens a stream for `dat`, then: (1) no new stream for `dat` is opened without closing the current stream, and (2) any open stream for this file must be closed by the time the top-level call to `foo` returns." Such a discipline follows programmer intuition and prevents security flaws where the main context, unaware that `foo` has left open a sensitive stream, invokes an untrusted program that can now do I/O on the "leaked" stream (for a real instance of such a bug, see Section 5.4).

A PAL monitor for this requirement is shown in Figure 5.3. The states of the monitor are encoded by a set of *monitor variables*, and its transitions by a set of `event{...}` blocks. Some monitor variables are *global* and are declared using the keyword `global` — intuitively, global monitor variables may be tested or updated by any event. In addition, each event includes an optional set of *local monitor variables*, declared using the keyword `local`, whose scope is restricted to the current event.

Events are fired by matching *patterns* on statements in the analyzed program. A pattern, specified in a `pattern{...}` block, is an assignment or procedure call with possible "pattern variables" ($?, $1, $2, etc.). During matching, the variables $1, $2, etc. match arbitrary C expressions and the variable $? serves as a wildcard— e.g., the pattern in event 1 matches all calls to `fopen`. For each statement matching the pattern [1] specified in the i-th event, the monitor sets up a precondition and a postcondition using the code in the blocks `before{...}` and `after{...}` in this event. The precondition (similarly, postcondition) checks whether an optional guard—a C

[1]Monitors are deterministic—i.e., if more than one pattern is matched at any point, we break the tie by picking the one in the event defined first.

expression over monitor and pattern variables, inside a `guard{...}` block—is satisfied by the monitor state right before (after) this statement. If the guard is not satisfied, an assertion violation is reported. Otherwise, the state of the monitor is updated by executing the C code contained within an optional `action{...}` block. This code is allowed to read pattern variables, and read or update monitor variables. For succinctness, we allow guards and actions to be defined outside `before` or `after` blocks (event 1 or 2)—in this case they are assumed to define preconditions.

During instrumentation, code blocks implementing an event's precondition and postcondition are respectively injected before and after statements matching its pattern. Consider a call `x = foo(y)` in a procedure `bar` in a program; on instrumentation using the monitor in Figure 5.3, this line is replaced by the chunk of code in Figure 5.4. Declarations of the monitor variables are added as well; `stored` is declared locally in `bar`, and `infoo`, `open`, and `stream` are declared globally.

Note that this syntax closely resembles that of the Blast query language. Blast, too, allows injection of code before or after a program statement using the keywords `before` and `after`. This similarity is a design feature, as our goal was to extend Blast minimally to obtain a specification language for context-sensitive requirements. The key new features in Pal are local variables and the ability to declare `before` *and* `after` blocks in the same event. This modification makes a major semantic difference: the control-flow of a monitor is now given by a nested word automaton, rather than a word automaton. Consider our example monitor and an execution of the input program containing a call to `foo`. In the nested word capturing this execution, there is a jump-edge from the call to `foo` to its matching return. Now, as the monitor reads this execution, it can save its state right before control enters `foo` using its local variables, and retrieve this state at the matching return. Thus, it has the power of a nested word monitor that reads the corresponding nested word, consulting its state at the source of an incoming jump-edge while transitioning to a return position. On the other hand, our monitor can use its global variables to pass states *into* invoked procedures such as `foo`, just like a BLAST monitor. More abstractly, this amounts to state updates as it reads the underlying word structure.

We end this section with some hints to check that the monitor in Fig. 5.3 specifies our original requirement. The variables `infoo` and `open` track whether `foo` is in the stack and whether `dat` is open, and `stream` stores a possible open stream for `dat`. The variable `stored` is used to infer whether control is back to the top-level context calling `foo`. The rest is easily verified.

5.4 Implementation and case studies

We have implemented PAL on top of the current implementation of BLAST. The specification and analysis modules in BLAST are orthogonal: the former generates C code instrumented with a monitor, while the latter checks the generated code for assertion failures. We extend BLAST's specification module to permit PAL monitors, and analyze the generated code statically as well as dynamically. The source code of our implementation, along with the examples that we now discuss, is available at `http://www.cis.upenn.edu/~swarat/tools/pal.tar.gz`.

File descriptor leak in `fcron`

We can apply a monitor as in Section 5.2 to prevent a reported bug in Version 2.9.4 of `fcron`, a periodic command scheduler for Linux. Here, the main function of a binary (`fcrontab`) calls a routine `parseopt`, which calls a routine `is_allowed` to check if a user is "allowed", which calls a procedure that opens, but forgets to close, a stream for a secret file `fcron.allow`. After control returns to the main context, the program starts a process with a name derived from an environment variable. However, an attacker can change the value of this variable to start a malicious program that reads `fcron.allow` via the open file stream.

This error may be prevented by a policy that allows `parseopt` to open `fcron.allow`, but not to leak its descriptor. Also, we could require that this secret file is not opened outside the scope of `parseopt`. This policy makes intuitive sense: as `parseopt` is a routine verifying a username, it is reasonable that it, or procedures it calls transitively, opens the file of allowed users. However, by the principle of least privilege, this file should only be opened when necessary, i.e., when `parseopt` is on the stack. A monitor expressing these requirements looks very similar to the one in Fig. 5.3. On instrumenting `fcron` with this monitor, we find a policy violation within a few random tests. However, abstraction-based model checking using BLAST is not suitable for this example, as BLAST cannot currently perform good analysis of library functions like `strcmp`.

Stack-sensitive security properties

Consider the security property: "A program must not execute a sensitive operation `write` at any point when an untrusted routine `foo` is on the stack." Note that this is the same property we discussed earlier. In the Java and C# languages, such policies are automatically enforced by the run-time environment, using the mechanism of *stack inspection*. In C, they may be enforced dynamically using a monitor—however, traditional monitors cannot express such properties of the stack, so that a nested word monitor is needed. Of course, such monitors could also be used in static analysis or software model checking.

Fig. 5.5 shows a complete monitor for this property. The global variable `infoo` tracks if `foo` is in the stack, and a guard prevents writes within the scope of `foo`.

We note that PAL may also be used to state some requirements of this nature that *cannot* be enforced via stack inspection. Consider the property: "If an untrusted procedure has ever been on the stack, a certain sensitive operation must not be executed." The rationale is that an untrusted routine may cause a side-effect that proves to be dangerous at a future point, so that if we call one, we must strengthen the security policy. However, since the culpable routine may no longer be on the stack when a violation occurs, stack inspection does not help in this case. On the other hand, it is easy to state such properties in PAL.

Logging policies

PAL also finds use in stating *logging policies* enforced in large development efforts such as Windows. Consider the property: "Whenever a procedure returns an error value, the error must be logged via a routine `log` before control leaves the current procedural context." Now, different development groups may call `log` via different wrapper functions; however, the logging policy is fixed across groups and thus independent of the wrappers. In order to track if control has returned from a wrapper to the original context, we need a PAL monitor.

While we do not have access to industrial code bases where such policies are most natural, we have applied a PAL monitor for this property on a couple of hand-coded examples. The monitors are quite simple and similar to our previous examples; hence we omit the details.

```
global int infoo = 0;
global int open = 0;
global FILE * stream;

event {  /* event 1 */
 pattern {
   $1=fopen($2,$?);}
 guard { strcmp($2,''dat'')
   ||(open==0 && infoo==1)}
 action {
   if (!strcmp($2,''dat'')){
     open = 1;
     stream = $1;}}
}

event {  /* event 2 */
  pattern { fclose ($1); }
  action { if ($1 == stream)
             open = 0; }}

event {  /* event 3 */
  pattern { $? = foo ($?); }
  local int stored;
  before {
    action { stored = infoo;
             infoo = 1; }}
  after {
    guard { open == 0 }
    action {
      infoo = stored; }}
}
```

Figure 5.3: A Pal specification

```
int infoo = 0;
int open = 0;
FILE * stream;

bar() {
  int stored;
  ...
  stored = infoo;
  infoo = 1;
  x = foo(y);
  if (open == 0)
     infoo = stored;
  else ERROR;
  ...
}
```

Figure 5.4: Instrumenting using event 3

```
global int infoo = 0;

event {
  pattern { write(); }
  guard { infoo == 0 }
}

event {
  pattern {$? = foo($?);}
  local int stored;
  before {
    action { stored = infoo;
             infoo = 1; }}
  after {
    action {infoo = stored;}
  }
}
```

Figure 5.5: Stack-sensitive security

Part II

Algorithms

Chapter 6

Algorithms for context-sensitive reachability analysis

In this chapter, we present our results on algorithms for reachability analysis of context-sensitive program abstractions. We do not view these abstractions as generators of nested structures any longer— as a result, we redefine them as *recursive state machines* (RSMs) rather than nested state machines. The reachability problem for RSMs is equivalent to a graph problem called CFL-reachability. While it has been believed to be cubic for a long time, we adapt an algorithmic technique due to Rytter [Ryt85, Ryt83] to obtain an asymptotically subcubic algorithm for it. We also offer two subcubic algorithms for the reachability problem for two subclasses of PDSs—called *bounded-stack PDSs* and *hierarchical state machines*.

The chapter is organized as follows. Section 6.1 defines the three classes of RSMs that interest us, CFL-reachability, and the fast set data structure. Section 6.2 discusses reachability in general RSMs and CFL-reachability. In Section 6.3, we study reachability for bounded-stack RSMs, and Section 6.4 briefly examines reachability in hierarchical state machines.

6.1 Basics

Recursive state machines (RSMs), introduced by Alur et al. [ABE+05], are finite-state-machines that can call other finite-state-machines recursively. RSMs are equivalent to pushdown systems, and any solution for RSM-reachability can be translated to a solution the same complexity for pushdown systems. In this section, we define three variants of recursive state machines. We also review their connection with the context-free language reachability problem.

Recursive state machines

A *recursive state machine* (RSM) M is a tuple $\langle M_1, M_2, \ldots, M_k \rangle$, where each $M_i = \langle L_i, B_i, Y_i, En_i, Ex_i, \rightarrow_i \rangle$ is a *component* comprising:

- a finite set L_i of *internal states*;
- a finite set B_i of *boxes*;
- a map $Y_i : B_i \rightarrow \{1, 2, \ldots, k\}$ that assigns a component to every box;
- a set $En_i \subseteq L_i$ of *entry states* and a set $Ex_i \subseteq L_i$ of *exit states*;
- an edge relation $\rightarrow_i \subseteq (L_i \cup \mathit{Retns}_i \setminus Ex_i) \times (L_i \cup \mathit{Calls}_i \setminus En_i)$, where $\mathit{Calls}_i = \{(b, en) : b \in B_i, en \in En_{Y_i(b)}\}$ is the set of *calls* and $\mathit{Retns}_i = \{(b, ex) : b \in B_i, ex \in Ex_{Y_i(b)}\}$ the set of *returns* in M_i.

Note that an edge cannot start from a call or an exit state, and cannot end at a return or an entry state. We assume that for every distinct i and j, L_i, B_i, Calls_i, Retns_i, L_j, B_j, Calls_j, and Retns_j are pairwise disjoint. Arbitrary calls, returns and internal states in M are referred to as *states*. The set of all states is given by $V = \bigcup_i (L_i \cup \mathit{Calls}_i \cup \mathit{Retns}_i)$, and the set of states in M_j is denoted by V_j. We also write $B = \bigcup_i B_i$ to denote the collection of all boxes in M. Finally, the extensions

(a)

```
int g;              void bar()
                    {
void main ()         int y = 0;
{                   }
   int x = 1;
   bar();
   g = 1;
   bar();
L: x = g;
}
```

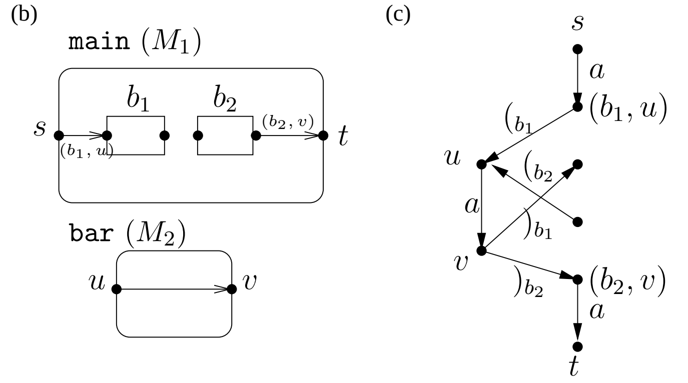

Figure 6.1: (a) A C example (b) RSM for the uninitialized variable problem (c) CFL-reachability formulation

of the relations $\rightarrow_i$ and functions Y_i are denoted respectively by $\rightarrow \subseteq V \times V$ and $Y : B \rightarrow \{1, 2, \ldots, k\}$.

For an example of an RSM, see Figure 6.1-(b). This machine has two components: M_1 and M_2. The component M_1 has an entry state s and an exit state t, boxes b_1 and b_2 satisfying $Y(b_1) = Y(b_2) = 2$, and edges $(s, (b_1, u))$ and $((b_2, v), t)$. The component M_2 has an entry u and an exit v, and an edge (u, v).

The semantics of M is given by an infinite *configuration graph* $\mathcal{C}_M$. Let a *configuration* of M be a pair $c = (v, w) \in V \times B^*$ satisfying the following condition: if $w = b_1 \ldots b_n$ for some $n \geq 1$ (i.e., if w is non-empty), then:

1. $v \in V_{Y(b_n)}$, and

2. for all $i \in \{1, \ldots, n-1\}$, $b_{i+1} \in B_{Y(b_i)}$.

The nodes of $\mathcal{C}_M$ are configurations of M. The graph has an edge from $c = (v, w)$ to $c' = (v', w')$ if and only if one of the following holds:

1. *Local move:* $v \in (L_i \cup \mathit{Retns}_i) \setminus \mathit{Ex}_i$, $(v, v') \in \rightarrow_i$, and $w' = w$;

2. *Call move:* $v = (b, en) \in \mathit{Calls}_i$, $v' = en$, and $w' = w.b$;

3. *Return move:* $v \in \mathit{Ex}_i$, $w = w'.b$, and $v' = (b, v)$.

Intuitively, the string w in a configuration (v, w) is a *stack*, and paths in $\mathcal{C}_M$ define the operational semantics of M. If v is a call (b, en) in the above, then the RSM pushes b on the stack and moves to the entry state en of the component $Y(b)$. Likewise, on reaching an exit ex, it pops a frame b off the stack and moves to the return (b, ex). Unsurprisingly, RSMs have linear translations to and from pushdown systems [ABE+05].

Size The *size* of an RSM is the total number of states in it.

Reachability Reachability in the configuration graph is defined as usual. We call the state v' *reachable* from the state v if a configuration (v', w), for some stack w, is reachable from (v, ϵ) in the configuration graph. Intuitively, the RSM, in this case, has an execution that starts at v with an empty stack and ends at v' with some stack. The state v' is *same-context reachable* from v if (v', ϵ) is reachable from (v, ϵ). In this case the RSM can start at v with an empty stack and reach v' with an empty stack—note that this can happen only if v and v' are in the same component.

The *all-pairs reachability problem* for an RSM is to determine, for each pair of states v, v', whether v' is reachable from v'. The single-source and single-sink variants of the problem are defined in the natural way. We also define the *same-context reachability problem*, where we ask if v' is same-context reachable from v.

All known algorithms for RSM-reachability and pushdown systems, whether all-pairs or single-source/single-sink, same-context or not, rely on a dynamic programming scheme called *summarization* [SP81, ABE$^+$05, BEM97, RHS95], which we will examine in Section 6.2. The worst-case complexity of all these algorithms is cubic. Tighter bounds are possible if we constrain the number of entry and exit states and/or edges in the input. For example, if each component of the input RSM has one entry and one exit state, then single-source, single-sink reachability can be determined in $O(m + n)$ time, where m is the number of edges in the RSM and n the number of states (the all-pairs problem has the same complexity as graph transitive closure) [ABE$^+$05]. In this paper, in addition to general RSM-reachability, we study reachability algorithms for RSMs constrained in a different way: *by restricting or disallowing recursion.*

To see the use of RSM-reachability in solving a program analysis problem, consider the program in Figure 6.1-(a). Suppose we want to determine if the variable `g` is uninitialized at the line labeled `L`. This may be done by constructing the RSM in Figure 6.1-(b). The two components correspond to the procedures `main` and `bar`; states in these components correspond to the control points of the program—e.g., the state s models the entry point of `main`, and (b_2, v) models the point immediately before line `L`. Procedure calls to `bar` are modeled by the boxes b_1 and b_2. For every statement that does not assign to `g`, an edge is added between the states modeling the control points immediately before and after this statement. Then `g` is uninitialized at `L` iff (b_2, v) is reachable from s. More generally, RSM-reachability algorithms can be used to check if a context-sensitive program abstraction satisfies a safety property [ABE$^+$05]. For example, the successful software model checker Slam [BR01] uses an algorithm for RSM-reachability as a core module.

Bounded-stack RSMs and hierarchical state machines

Now we define two special kinds of RSMs with restricted recursion: *bounded-stack RSMs* and *hierarchical state machines.* We will see later that they have better reachability algorithms than general RSMs.

The class of *bounded-stack RSMs* consists of RSMs M where every call (b, en) is unreachable from the state en. By the semantics of an RSM, the stack of an RSM grows along an edge from a call to the corresponding entry state. Thus, intuitively, a bounded-stack RSM forbids infinite recursive loops, ensuring that in any path in the configuration graph starting with a configuration (v, ϵ), the height of the stack stays bounded. To see an application, consider a procedure that accepts a boolean value as a parameter, flips the bit, and, if the result is 1, calls itself recursively. While this program does employ recursion, it never runs into an infinite recursive loop. As a result, it can be modeled by a bounded-stack RSM.

A *hierarchical state machine* [AY98], on the other hand, forbids recursion altogether. Formally, such a machine is an RSM M where there is a total order $\prec$ on the components $M_1, \ldots, M_k$ such that if M_i contains a box b, then $M_{Y(b)} \prec M_i$. Thus, calls from a component may only lead to a component lower down in this order. For example, the RSM in Figure 6.1-(b) is a hierarchical state machine.

Note that every bounded-stack or hierarchical machine can be translated to an equivalent finite-state machine. However, this causes an exponential increase in size in the worst case, and it is unreasonable to analyze a hierarchical/bounded-stack machine by "flattening" it into a finite-state machine. The question that interests us is: can we determine reachability in a bounded-stack or hierarchical machine in time *polynomial in the input*? The only known way to do this is to use the summarization technique that also works for general RSMs, leading to an algorithm of cubic worst-case complexity.

Context-free language reachability

RSM-reachability is equivalent to a graph problem called *context-free language (CFL) reachability* [Yan90, Rep98] that has numerous applications in program analysis. Let S be a directed graph whose edges are labeled by an alphabet Σ, and let L be a context-free language over Σ. We say a node t is L-reachable from a node s if there is a path from s to t in S that is labeled by a word in L. The all-pairs CFL-reachability problem for S and L is to determine, for all pairs of nodes s and t, if t is L-reachable from s. The single-source or single-sink variants of the problem are defined in the obvious way. Customarily, the size of the instance is given by the number n of nodes in S, while L is assumed to be given by a fixed-sized grammar G.

Let us now see how, given an instance of RSM-reachability, we can obtain an equivalent CFL-reachability instance. We build a graph whose nodes are states of the input RSM M; for every edge (u, v) in M, S has an edge from u to v labeled by a symbol a. For every call (b, en) in the RSM, S has an edge labeled $(_b$ from (b, en) to en; for every exit ex and return (b, ex) in M, we add a $)_b$-labeled edge in S from ex to (b, ex) (for example, the graph S constructed from the RSM in Figure 6.1-(b) is shown in Figure 6.1-(c)). Now, the state v is reachable from the state u in M if and only if v is L-reachable from u in S, where L is given by the grammar $S \rightarrow SS \mid (_b S)_b \mid (_b S \mid a$. The translation in the other direction is also easy—we refer the reader to the original paper on RSMs [ABE+05].

Note that context-free recognition is the special case of CFL-reachability where S is a chain. A cubic algorithm for all-pairs CFL-reachability can be obtained by generalizing the Cocke-Younger-Kasami algorithm [HU79] for CFL-recognition—this algorithm again relies on summarization. The problem is known to be equivalent to the problem of evaluating Datalog *chain queries* on a graph representation of a database [Yan90]. Such queries have the form $p(X, Y) \leftarrow q_0(X, Z_1) \wedge q_1(Z_1, Z_2) \wedge \ldots \wedge q_k(Z_k, Y)$, where the q_i's are binary predicates and X, Y and the Z_i's are distinct variables, and have wide applications. It has also come up often in program analysis—for example, in the context of interprocedural dataflow analysis and slicing, field-sensitive alias analysis, and type-based flow analysis [HRB88, RHS95, HRS95, Rep95, Rep98, RF01]. The "cubic bottleneck" of these analysis problems has sometimes been attributed to the believed cubic hardness of CFL-reachability.

A special case is the problem of *Dyck-CFL-reachability*. The constraint here is that the CFL L is now a language of balanced parentheses. Many program analysis applications of CFL-reachability—e.g., field-sensitive alias analysis of Java programs [SGSB05]—turn out actually to be applications of Dyck-CFL-reachability, though so far as asymptotic bounds go, it is no simpler than the general problem. This problem is equivalent to the problem of same-context reachability in RSMs.

Fast sets

Our algorithms for RSMs use a set data structure that exploits sharing between sets to offer certain set operations at low amortized cost. This data structure—called *fast sets* from now on—is standard technology in the algorithms literature [Cha07, ADKz70] and was used, in particular, in the papers by Rytter [Ryt83, Ryt85] on two-way pushdown recognition. Its essence is that it splits an operation on a pair of sets into a series of unit-cost operations on small sets. We will now review it.

Let U be a universe of n elements of which all our sets will be subsets. The fast set data structure supports the following operations:

- *Set difference*: Given sets X and Y, return a *list* $\mathit{Diff}(X, Y)$ consisting of the elements of the set $(X \setminus Y)$.

- *Insertion*: Insert a value into a set.

- *Assign-union*: Given sets X and Y, perform the assignment $X \leftarrow X \cup Y$.

Let us assume an architecture with word size $p = \theta(\log n)$. A fast set representation of a set is the bit vector (of length n) for the set, broken into $\lceil n/p \rceil$ words. Then:

- To compute $\mathit{Diff}(X, Y)$, where X and Y are fast sets, we compute the bit vector for $Z = X \setminus Y$ via bitwise operations on the words comprising X and Y. This takes $O(n/p)$ time assuming constant-time logical operations on words. To list the elements of Z, we repeatedly locate the most significant bit in Z, add its position in X to the output list, and turn it off. Assuming that it is possible in constant time to check if a word equals 0 and find the most significant bit in a word, this can be done in $O(|Z| + n/p)$ time. Note that the bound is given in terms of the *size of the output*. This is exploited while bounding the amortized cost of a sequence of set differences.

- Insertion of $0 \leq x \leq n-1$ involves setting a bit in the $\lfloor x/p \rfloor$-th word, which can be done in $O(1)$ time.

- The assign-union operation can be implemented by word-by-word logical operations on the components of X and Y, and takes $O(n/p)$ time.

In case the unit-cost operations we need are not available, they can be implemented using table lookup. Let a fast set now be a collection of words of length $p = \lceil \log n/2 \rceil$. In a preprocessing phase, we build tables implementing each of the binary or unary word operations we need by simply storing the result for each of the $O(2^p.2^p) = O(n)$ possible inputs. The time required to build each such table is $O(p.n)$ (assuming linear-time operations on words), and the space requirement is $O(n)$. The costs of our fast set operations are now as before.

6.2 All-pairs reachability in recursive state machines

Let us now study the reachability problem for recursive state machines. We remind the reader that all known algorithms for this problem are cubic and based on a high-level algorithm called *summarization*. In this section we show that a speedup technique developed by Rytter [Ryt85, Ryt83] can be directly applied to this algorithm, leading to an $O(n^3/\log n)$-time solution. The modified algorithm computes reachability via a sequence of operations on *sets of states*, each represented as a fast set. In this sense it is a *symbolic* implementation of summarization, rather than an iterative one like the popular algorithm due to Reps, Horwitz and Sagiv [RHS95].

We also show that the standard cubic algorithm for CFL-reachability, referenced for example by Melski and Reps [MR00], can be speeded up similarly using Rytter's technique.

6.2.1 Reachability in RSMs

Let us start by reviewing summarization. We have as input an RSM $M = \langle M_1, \ldots, M_k \rangle$ as in Section 6.1, with state set V, box set B, edge relation $\rightarrow \subseteq V \times V$, and a map $Y : B \rightarrow \{1, \ldots, k\}$ assigning components to boxes. The algorithm first determines *same-context reachability* by building a relation $H^s \subseteq V \times V$, defined as the least relation satisfying:

1. if $u = v$ or $u \rightarrow v$, then $(u, v) \in H^s$;
2. if $(u, v') \in H^s$ and $(v', v) \in H^s$, then $(u, v) \in H^s$;
3. if $(u, v) \in H^s$ and u is an entry and v is an exit in some component, then for all boxes b such that $(b, u), (b, v) \in V$, we have $((b, u), (b, v)) \in H^s$.

For example, the relation H^s for the RSM in Figure 6.1-(a) is drawn in Figure 6.2 (the transitive edges are omitted). While the definition of H^s is recursive, it may be constructed using a least-fixpoint computation. Once it is built, we construct a relation $H \subseteq V \times V$ defined as:

$$\begin{aligned} H \;=\; \rightarrow \;\cup\; &\{((b, en), (b, ex)) \in H^s : b \in B, \text{ and } en \text{ is an} \\ &\quad \text{entry and } ex \text{ an exit of } Y(b)\} \\ \cup\; &\{((b, en), en) : en \text{ is an entry in } Y(b)\}, \end{aligned}$$

and compute the (reflexive) transitive closure H^* of the resultant relation (see Figure 6.2). It is known that:

Lemma 8 ([ABE+05, BEM97]). *For states v and v' of M, v' is reachable from v iff $(v, v') \in H^*$. Also, v' is same-context reachable from v iff $(v, v') \in H^s$.*

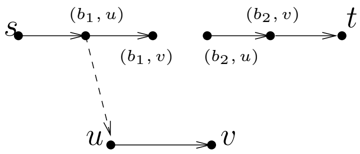

Figure 6.2: The relation H. H^s is the transitive closure of non-dashed edges, and H^* is the transitive closure of all edges

Within the scheme of summarization, there are choices as to how the fixpoint computations for H^s and H^* are carried out. For example, the popular algorithm due to Reps, Horwitz and Sagiv [RHS95] employs graph search to construct these relations enumeratively. In contrast, the algorithm we now present, obtained by a slight modification of an algorithm by Rytter [Ryt85] for two-way pushdown recognition, phrases the computation as a sequence of operations on *sets of states*. Unlike previous implementations of summarization, our algorithm has a slightly subcubic worst-case complexity.

The algorithm is a modification of the procedure Baseline-Reachability in Figure 6.3, which uses a worklist W to compute H^s and H^* in a fairly straightforward way. Line 1 of the baseline routine inserts intra-component edges and trivial reachability facts into H^s and W. The rest of the pairs in H^s are derived by the while-loop from line 2–10, which removes pairs from W one by one and "processes" them. While processing a pair (u, v), we derive all the pairs that it "implies" by rules (2) and (3) in the definition of H^s and *that have not been derived already*, and insert them into H^s and W. At the end of any iteration of the loop, W contains the pairs that have been derived but not yet processed. The loop continues till W is empty. It is easy to see that on its termination, H^s is correctly computed. Lines 11-14 now compute H^*.

Note that a pair is inserted into W only when it is also inserted into H^s, so that the loop has one iteration per insertion into H^s. At the same time, a pair is never taken out of H^s once it is inserted, and no pair is inserted into it twice. Let n be the size of the RSM, and let $\alpha \leq n^2$ be an upper bound on the number of pairs (u, v) such that v is reachable from u. Then the loop has $O(\alpha)$ iterations.

Let us now determine the cost of each iteration. Assuming we can insert an element in H^s and W in constant time, lines 4–6 cost constant time per insertion of an element into H^s. Thus, the total cost for lines 4–6 during a run of Baseline-Reachability is $O(\alpha)$. The for-loops at line 7 and line 9 need to identify all states u' and v' satisfying their conditions for insertion. Done enumeratively, this costs $O(n)$ time per iteration, causing the total cost of the loop to be $O(\alpha n)$. As for the rest of the algorithm, line 14 may be viewed as computing the (reflexive) transitive closure of a graph with n states and $O(\alpha)$ edges. This may clearly be done in $O(\alpha n)$ time. Then:

Lemma 9. Baseline-Reachability *terminates on any RSM M in time $O(\alpha.n)$, where $\alpha \leq n^2$ is the number of pairs $(u, v) \in V \times V$ such that v is reachable from u. On termination, for every pair of states u and v, v is reachable from u iff $(u, v) \in H^*$, and v is same-context reachable from u iff $(u, v) \in H^s$.*

To convert the baseline procedure into a set-based algorithm, interpret the relation H^s as an $n \times n$ table, and denote the u-th row and column as sets (respectively denoted by $Row(u)$ and $Col(u)$). Then we have $Row(u) = \{v : (u, v) \in H^s\}$ and $Col(u) = \{v : (v, u) \in H^s\}$. Now observe that the for-loops at lines 7 and 9 can be captured by *set difference operations.* The for-loop in line 7–8 may be rewritten as:

$$\textbf{for } u' \in (Col(u) \setminus Col(v)) \textbf{ do } \text{ insert } (u', v) \text{ into } H^s \text{ and } W,$$

and the for-loop in line 9–10 may be rewritten as:

$$\textbf{for } v' \in (Row(v) \setminus Row(u)) \textbf{ do } \text{ insert } (u, v') \text{ into } H^s \text{ and } W.$$

Our set-based algorithm for RSM-reachability —called Reachability from now on— is obtained by applying these rewrites to Baseline-Reachability. Clearly, Baseline-Reachability()

$W \leftarrow H^s \leftarrow \{(u,u) : u \in V\} \cup \rightarrow$
while $W \neq \emptyset$
do $(u,v) \leftarrow$ remove from W
  if u is an entry state and v an exit state in a component M_i
    then for b such that $Y(b) = i$
      do insert $((b,u),(b,v))$ into H^s, W
  for $(u',u) \in H^s$ such that $(u',v) \notin H^s$
  do insert (u',v) into H^s and W
  for $(v,v') \in H^s$ such that $(u,v') \notin H^s$
  do insert (u,v') into H^s and W
$H^* \leftarrow H^s$
for calls $(b, en) \in V$
do insert $((b,en), en)$ into H^*
$H^* \leftarrow$ transitive closure of H^*

Figure 6.3: Baseline procedure for RSM-reachability

Reachability terminates after performing $O(\alpha)$ set difference and insertion operations, and when it does, the tables H^* and H^s respectively capture reachability and same-context reachability.

We may, of course, use any set data structure offering efficient difference and insertion in our algorithm. If the cost of set difference is linear, then the algorithm is cubic in the worst-case. The complexity, however, becomes $O(n\alpha/\log n) = O(n^3/\log n)$ if we use the fast set data structure of Section 6.1. To see why, assume that the rows and columns of H^s are represented as fast sets and that set difference and insertion are performed using the operations *Diff* and *Ins* described earlier. In each iteration of the main loop, the inner loops first compute the difference of two sets of size n, then, for every element in the answer, inserts a pair into H^s (this involves inserting an element into a row and a column) and W. If the i-th iteration of the main loop inserts σ_i pairs into H^s, the time spent on the operation *Diff* in this iteration is $O(n/\log n + \sigma_i)$. Since the result is returned as a list, the cost of iteratively inserting pairs in it into H^* and W is also $O(\sigma_i)$. The cost of these operations summed over the entire run of Reachability is $O(\alpha.n/\log n + \sum_i^\alpha \sigma_i) = O(\alpha n/\log n + \alpha) = O(\alpha n/\log n)$.

The only remaining bottleneck is the transitive closure in line 14 of the baseline procedure. This may be computed in $O(\alpha.n/\log n)$ time using the procedure we give in Section 6.3.1. The total time complexity then becomes $O(\alpha n/\log n)$— i.e., $O(n^3/\log n)$.

As for the space requirement of the algorithm, $\Theta(n^2)$ space is needed just to store the tables H^s and H^*. The space required by tables implementing word operations, if unit-cost word operations are not available, is subsumed by this factor. Thus we have:

Theorem 20. *The algorithm* Reachability *solves the all-pairs reachability and same-context-reachability problems for an RSM with n states in $O(n^3/\log n)$ time and $O(n^2)$ space.*

Readers familiar with Rytter's $O(n^3/\log n)$-time algorithm [Ryt85] for recognition of two-way pushdown languages will note that our subcubic algorithm is very similar to it. Recall that a two-way pushdown automaton (2-PDA) is a pushdown automaton which, on reading a symbol, can move its "reading head" one step forward and back on the input word, while changing its control state and pushing/popping a symbol on/off its stack. The language recognition problem for 2-PDAs is: "given a word w of length n and a 2-PDA $\mathcal{A}$ of constant size, is w accepted by $\mathcal{A}$?" This problem may be linearly reduced to the reachability problem for RSMs. Notably, there is also a reduction in the other direction. Given an RSM M where we are to determine reachability, write out the states and transitions of M as an input word. Now construct a 2-PDA $\mathcal{A}$ that, in every one of an arbitrary number of rounds, moves its head to an arbitrary transition of M and tries to simulate the execution. Using nondeterminism, $\mathcal{A}$ can guess any run of M, and accept the input if and only if M has an execution from a state u to a state v. This may suggest that a subcubic algorithm for RSM-reachability already exists. The catch, however, is that an RSM of size n may have $\Omega(n^2)$ transitions, so that this reduction outputs an instance of quadratic size. Clearly, it cannot be combined with Rytter's algorithm to solve reachability in RSMs in cubic (let alone subcubic) time.

On the other hand, what Rytter's algorithm actually does is to speed up a slightly restricted form of summarization. Recall the routine Baseline-Reachability, and let $u, v, \ldots$ be positions in a word rather than states of an RSM. Just like us, Rytter derives pairs (u, v) such that the automaton has an empty-stack to empty-stack execution from u to v. One of the rules he uses is:

> Suppose (u, v) is already derived. If $\mathcal{A}$ can go from u' to u by pushing γ, and from v to v' by popping γ, then derive (u', v').

This rule is analogous to Rule (3) in our definition of summarization:

> Suppose (u, v) is already derived. If u is an entry and v is an exit in some component and b is a box such that $(b, u), (b, v) \in V$, then derive $((b, u), (b, v))$.

The two rules differ in the number of new pairs they derive. Because the size of $\mathcal{A}$ is fixed, Rytter's rule can generate at most a constant number of new pairs for a fixed pair (u, v). On the contrary, our rule can derive a linear number of new pairs for given (u, v). Other than the fact that Rytter deals with pairs of positions and we deal with RSM states, this is the only point of difference between the baseline algorithms used in the two cases. At first glance, this difference may seem to make the algorithm cubic, as the above derivation happens inside a loop with a quadratic number of iterations. Our observation is that a tighter analysis is possible: our rule above only does a constant amount of work *per insertion* of a pair into H^s. Thus, over a complete run of the algorithm, its cost is quadratic and subsumed by the cost of the other lines, even after the speedup is applied. For the rest of the algorithm, Rytter's complexity arguments carry over.

```
Baseline-CFL-Reachability()
W ← H^s ← {(u, A, v) : u --a--> v in S, and  A → a in G }
               ∪{(u, A, u) :  A → ε in G }
while W ≠ ∅
do (u, B, v) ←  remove from W
    for  each production A → B
    do  insert (u, B, v) into H^s, W
    for  each production A → CB
    do for  each edge (u′, C, u) such that (u′, A, v) ∉ H^s
        do  insert (u′, A, v) into H^s and W
    for  each production A → BC
    do for  each edge (v, C, v′) such that (v, A, v′) ∉ H^s
        do  insert (v, A, v′) into H^s and W
```

Figure 6.4: Baseline algorithm for CFL-reachability

6.2.2 CFL-reachability

As RSM-reachability and CFL-reachability are equivalent problems, the algorithm Reachability can be translated into a set-based, subcubic algorithm for CFL-reachability. However, Rytter's technique can also be directly applied to the standard algorithm for CFL-reachability, described for example by Melski and Reps [MR00]. Now we show how. Let us have an instance (S, G) of CFL-reachability, where S is an edge-labeled graph with n nodes and G is a constant-sized context-free grammar. Without loss of generality, it is assumed that the right-hand side of each rule in G has at most two symbols. The algorithm in Melski and Reps' paper—called Baseline-CFL-Reachability and shown in Figure 6.4—computes tuples (u, A, v), where u, v are nodes of S and A is a terminal or non-terminal, such that there is a path from u to v labeled by a word w that G can derive from A. A worklist W is used to process the tuples one by one; derived tuples are stored in a table H^s. It is easily shown, by arguments similar to those for RSM-reachability, that the algorithm is cubic and requires quadratic space. On termination, a tuple (u, I, v), where u, v are nodes and I the initial symbol of G, is in H^s iff v is CFL-reachable from u.

As in case of RSM-reachability, now we store the rows and columns of H^s as fast sets of $O(n)$ size. For a node u and a nonterminal A, the row $Row(u, A)$ (similarly the column $Col(u, A)$), stores the set of nodes u' such that (u, A, u') (similarly (u', A, u)) is in H^s. Now, the bottlenecks of the algorithm are the two nested loops (lines 7–9 and 10–12). We speed them up by implementing them using set difference operations— for example, the loop from line 7–9 is replaced by:

for each production $A \to CB$
do for $u' \in (Col(u, C) \setminus Col(v, A))$
  do insert (u', A, v) into H^s and W.

Assuming a fast set implementation, the cost for this loop is in a given iteration of the main loop is $O(n/\log n + \sigma)$, where σ is the number of new tuples inserted into H^s. Since the number of insertions into H^s is $O(n^2)$, its total cost during a complete run of the algorithm is $O(n^3/\log n)$. The same argument holds for the other loop. Let us call the modified algorithm CFL-Reachability. By the discussion above:

Theorem 21. *The algorithm* CFL-Reachability *solves the all-pairs CFL-reachability problem for a fixed-sized grammar and a graph with n nodes in $O(n^3/\log n)$ time and $O(n^2)$ space.*

Theorem 21 improves the previous cubic bound for all-pairs— or, for that matter, single-source, single-sink— CFL-reachability. By our discussion in Section 6.1, this implies subcubic, set-based algorithms for Datalog chain query evaluation as well as the many program analysis applications of CFL-reachability.

6.3 All-pairs reachability in bounded-stack RSMs

Is a better algorithm for RSM-reachability possible if the input RSM is bounded-stack? In this section, we show that this is indeed the case. As we mentioned earlier, the only previously known way to solve reachability in bounded-stack machines is to use summarization, which gives a cubic algorithm; speeding it up using the technique we presented earlier leads to a factor-$\log n$ speedup. Now we show that the bounded-stack property gives us a second logarithmic speedup. Our algorithm combines graph search with a speedup technique used by Rytter [Ryt83, Ryt85] to recognize languages of loop-free 2-way PDAs[1]. Unlike the algorithm for general RSMs, it is not just an application of existing techniques, and we consider it the main new algorithm of this paper.

We start by reviewing search-based algorithms for reachability in (general) RSMs. Let M be an RSM as in Section 6.1, and recall the relation H defined in Section 6.2—henceforth, we view it as a graph and call it the *summary graph* of M. The edges of H are classified as follows:

- Edges $((b, en), en)$, where b is a box and en is an entry state in $Y(b)$, are known as *call edges*;
- Edges $((b, en), (b, ex))$, where b is a box, and en is an entry and ex an exit in $Y(b)$, are called *summary edges*;
- Edges that are also edges of M are called *local edges*.

Note that a state v is same-context reachable from a state u iff there is a path in H from u to v made only of local and summary edges. Let the set of states same-context reachable from u be denoted by $H^s(u)$. While the call and local edges of H are specified directly by M, we need to determine reachability between entries and exits in order to identify the summary edges. The search-based formulation of summarization [RHS95, HRS95] views reachability computation for M (or, in other words, computation of the transitive closure H^* of H) as a restricted form of *incremental* transitive closure. A search algorithm is employed to compute reachability in H; when an exit ex is found to be same-context-reachable from en, the summary

[1] A loop-free 2-PDA is one that has no infinite execution on any word. The recognition problem for loop-free 2-PDAs reduces to reachability in *acyclic* RSMs—i.e., RSMs whose configuration graphs are cycle-free. Obviously, these are less general than bounded-stack RSMs.

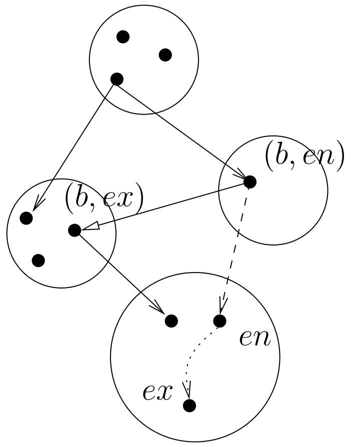

Figure 6.5: All-pairs reachability in bounded-stack RSMs

edge $((b, en), (b, ex))$ is *added to the graph.* The algorithm must now explore these added edges along with the edges in the original graph.

Let us now assume that M is bounded-stack. Consider any call (b, en) in the summary graph H. Because M is bounded-stack, this state is unreachable from the state en. Hence, (b, en) and en are not in the same strongly connected component (SCC) in H, and a call edge is always between two SCCs. The situation is sketched in Figure 6.5. The nodes are states of M (en is an entry and ex is an exit in the same component, while b is a box), and the large circles denote SCCs. We do not draw edges within the same SCC—the dotted line from en to ex indicates that ex is same-context reachable from en.

We will argue that all summary edges in H may be discovered using a variant of depth-first graph search (DFS). To start with, let us assume that the summary graph H is acyclic, and consider a call (b, en) in it. First we handle the case when no path in H from en contains a call. As a summary-edge always starts from a call, this means that no such path contains a summary-edge either, and the part of H reachable from en is not modified due to summary edge discovery. Thus, the set $H^s(en)$ of states v same-context reachable (i.e., reachable via summary and local edges) from en can be computed by exploring H depth-first from en. Further, because the graph is acyclic, the same search can label each such v with the set $H^s(v)$. This is done as follows:

- if v has no children, then $H^s(v) = \{v\}$;
- if v has children $u_1, u_2, \ldots, u_m$, then

$$H^s(v) = \bigcup_i H^s(u_i).$$

Once we have computed the set $H^s(en)$ of such v-s that are same-context reachable from en, we can, consulting the transition relation of M, determine all summary edges $((b, en), (b, ex))$. Note that these are the only summary edges from (b, en) that can ever be added to H. However, these summary edges may now be explored via *the same depth-first traversal*—we may view them simply as edges explored after the call-edge to en due to the DFS order. The same search can compute the set $H^s(u)$ for each new state u found to be reachable from the return (b, ex). Note that descendants of (b, ex) may also be descendants of en—for example, a descendant x of en may be reachable from a different entry point en' of $Y(b)$, which may be "called" by a call reachable from (b, ex). In other words, the search from (b, ex) may encounter some *cross-edges*, thus needing to use some of the H^s-sets computed during the search from en. Once the H^s-sets for en and all summary-children (b, ex) are computed, we can compute the set $H^s((b, en))$. Since we are only interested in reachability via summary and local edges and a call has no local out-edges, this set is the union of the H^s-sets for the summary children.

Now suppose there are at most $p \geq 1$ call states in a path in H from en. Let the state (b', en') be the first call reached from en in a depth-first exploration— because of the bounded-stack property, no descendant of en' can reach en in H. Now, there can be at most $(p - 1)$ calls in a path from en', so that can inductively determine the summary edges from (b', en'), explore these edges, and label every state v in the resultant tree by the set $H^s(v)$. It is easy to see that this DFS can be "weaved" into the DFS from en.

The above algorithm, however, will not work when H has cycles. This is because in a graph with cycles, a simple DFS cannot construct the sets $H^s(v)$ for all states v. This difficulty, however, may be resolved if we use, instead of a plain DFS, a transitive closure algorithm based on Tarjan's algorithm to compute the SCCs of a graph [AHU74]. Many such algorithms are known in the literature [Pur70, EKS77, Sch83]. Let $\mathit{Reach}(v)$ denote the set of nodes reachable from a node v in a graph. The first observation that these algorithms use is that for any two nodes v_1 and v_2 in the same SCC of a graph, we have $\mathit{Reach}(v_1) = \mathit{Reach}(v_2)$. Thus, it is sufficient to compute the set *Reach* for a single representative node per SCC. The second main idea is based on a property of Tarjan's algorithm. To understand it, we will have to define the *condensation graph* $\widehat{G}$ of a graph G:

- the nodes of $\widehat{G}$ are the SCCs of G;
- the edge set is the least set constructed by: "if, for nodes S_1 and S_2 of $\widehat{G}$, G has nodes $u \in S_1, v \in S_2$ such that there is an edge from u to v, then $\widehat{G}$ has an edge from S_1 to S_2."

Now, Tarjan's algorithm, when running on a graph G, "piggybacks" a depth-first search of the graph and outputs the nodes of $\widehat{G}$ in a bottom-up topological order. This is possible because the condensation graph of any graph is acyclic. For example, running on the graph in Figure 6.5 (let us assume that all the edges are known), the algorithm will first output the SCC containing *en*, then the one containing (b, ex), then the one containing (b, en), etc. We can, in fact, view the algorithm as performing a DFS on the condensation graph of G. In the same way as when our input graph was acyclic, we can now compute, for every node S in the condensation graph, the set of nodes $\mathit{Reach}(S)$ reachable from that SCC, defined as:

$$\mathit{Reach}(S) = \bigcup_{u \in S} \mathit{Reach}(u).$$

For each S, this set is known by the time the algorithm returns from the first node in S to have been visited in the depth-first search.

Assuming that we have a transitive closure algorithm of the above form, let us focus on bounded-stack RSMs again. Let us also suppose that we are only interested in same-context reachability. We apply the transitive closure algorithm to the graph H after modifying it in the two following ways. First, we ensure that the sets $Reach(u)$, for a state u, only contain descendants of u reachable via local and summary edges—this requires a trivial modification of the algorithm. To understand the second modification, consider once again a call (b, en) in a summary graph H; note that the call edge $((b, en), en)$ is an edge in the condensation graph $\widehat{H}$. Thus, the set $Reach(S_{en})$, where S_{en} is the SCC of en, is known by the time the transitive closure algorithm is done exploring this edge. Now we can construct all summary edges from (b, en) and *add them as outgoing edges* from (b, en), viewing them, as in the acyclic case, as normal edges appearing after the call-edge in the order of exploration. The set $Reach(S_{(b,en)})$ can now be computed.

By the time the above algorithm terminates, $Reach(S_u) = H^s(u)$ for each state u— i.e., we have determined all-pairs same-context reachability in the RSM. To determine all-pairs reachability, we simply insert the call edges into the summary graph, and compute its transitive closure. In fact, we can do better: with some extra book-keeping, it is possible to compute reachability in the same depth-first search used to compute same-context reachability (i.e., summary edges).

Next we present an algorithm for graph transitive closure that, in addition to being based on Tarjan's algorithm, also uses fast sets to achieve a subcubic complexity. Using the technique outlined above, we modify it into an algorithm for bounded-stack RSM-reachability of $O(n^3/\log^2 n)$ complexity.

6.3.1 Speeding up search-based transitive closure

The algorithm that we now present combines a Tarjan's-algorithm-based transitive closure algorithm (studied, for example, by Schmitz [Sch83] or Purdom [Pur70]) with a fast-set-based speedup technique used by Rytter [Ryt83, Ryt85] to solve the recognition problem for a subclass of 2-PDAs. While subcubic algorithms for graph transitive closure have been known for a long time, this is, so far as we know, the first algorithm that is based on graph traversal and yet runs in $O(n^3/\log^2 n)$ time. Both these features are necessary for an $O(n^3/\log^2 n)$-time algorithm on bounded-stack RSMs.

As in our previous algorithms, we start with a baseline cubic-time algorithm and speed it up using fast sets. This algorithm, called BASELINE-CLOSURE and shown in Figure 6.6, is simply a DFS-based transitive closure algorithm. Let us first see how it detects strongly connected components in a graph G. The main idea is that in any DFS tree of G, the nodes belonging to a particular SCC form a subtree. The node u_0 in an SCC S that is discovered first in a run of the algorithm is marked as the *representative* of S; for each node v in S, $rep(v)$ denotes the representative of S (in this case u_0). A global stack L supporting the usual push and pop operations is maintained; $height(L)$ gives the height of the stack at any given time. As soon as we discover a node, we push it on this stack—note that for any SCC, the representative is the first node to be on this stack. For every node u, $dfsnum(u)$ is the height of the stack when it was discovered, and $low(u)$ equals, once the search from u has returned, the minimum *dfsnum*-value of a node that a descendant of u in the DFS tree has an edge to. Now observe that if $low(u) = dfsnum(u)$ at the point when the search is about to return from a node u, then u is the representative of some SCC. We maintain the invariant that all the elements above and inclusive of u in the stack belong to the SCC of u. Before returning from u, we pop all these nodes and output them as an SCC. Nodes in SCCs already generated are stored in a set *Done*.

Now we shall see how to generate the set of nodes reachable from a node of G. Let S be an SCC of G; we want to compute the set $Reach(S)$ of nodes reachable from S. Consider the condensation graph $\widehat{G}$ of G, where S is a node. If S has no children in the graph, then $Reach(S) = S$; if it has children $S_1, S_2, \ldots, S_k$, then $Reach(S) = \bigcup_i Reach(S_i)$. Once this set is computed, we store it in a table T

VISIT(u)
1 add u to *Visited*
2 $push(u, L)$
3 $low(u) \leftarrow dfsnum(u) \leftarrow height(L)$
4 $T(u) \leftarrow \emptyset$; $rep(u) \leftarrow \perp$
5 $Out(u) \leftarrow \emptyset$; $Next(u) = \{$ children of u $\}$
6 **for** $v \in Next(u)$
7 **do if** $v \notin Visited$ **then** VISIT (v)
8   **if** $v \in Done$
9   **then** add v to $Out(u)$
10   **else** $low(u) \leftarrow \min(low(u), low(v))$
11 **if** $low(u) = dfsnum(u)$
12   **then repeat**
13             $v \leftarrow pop(L)$
14   add v to *Done*
15   add v to $T(u)$
16             $Out(u) \leftarrow Out(u) \cup Out(v)$
17             $rep(v) \leftarrow u$
18   **until** $v = u$
19       $T(u) \leftarrow T(u) \cup \bigcup_{v \in Out(u)} T(rep(v))$

BASELINE-CLOSURE()
1 $Visited \leftarrow \emptyset$; $Done \leftarrow \emptyset$
2 **for** each node u
3 **do if** $u \notin Visited$ **then** VISIT (u)

Figure 6.6: Transitive closure of a directed graph

indexed by the representatives of the SCCs of G.

Of course, we compute this set as well as generate the SCCs in one depth-first pass of G. Recall that the SCCs of G are generated in a bottom-up topological order (the outputting of SCCs is done by lines 12–19 of VISIT, the recursive depth-first traversal routine of our algorithm). By the time S is generated, the SCCs reachable from it in $\widehat{G}$ have all been generated, and the entries of T corresponding to the representatives of these reachable SCCs have been precisely computed. Then all we need to fill out $T(u_0)$, where u_0 is the representative of S, is to track the edges out of S and take the union of S and the entries of T corresponding to the children of S

in $\widehat{G}$. Note that these outgoing edges could either be edges in the DFS tree or DFS "cross edges." They are tracked using a table *Out* indexed by nodes of G—for any u in S, $Out(u)$ contains the nodes outside of S to which an edge from u may lead. At the end of the repeat-loop from line 13–18, $Out(u_0)$ contains all nodes outside S with an edge from inside S. Now line 19 computes the set of nodes reachable from u_0.

As for the time complexity of this algorithm, note that for each u, Visit(u) is called at most once. Every line other than 16 and 19 costs time $O(m + n)$ during a run of Baseline-Closure, and since line 16 tries to add a node to $Out(u)$ once for every edge out of the SCC of u in $\widehat{G}$, its total cost is $O(m)$. Line 19 does a union of two sets of nodes for each edge in $\widehat{G}$, so that its total cost is $O(mn)$. As for space complexity, the sets $Reach(u)$ can be stored using $O(n^2)$ space, a cost that subsumes the space requirements of the other data structures. Then we have:

Lemma 10. Baseline-Closure *terminates on any graph G with n nodes and m edges in time $O(mn)$. On termination, for every node u of G, $T(rep(u))$ is the set of nodes reachable from u. The algorithm requires $O(n^2)$ space.*

We will now show a way to speed up the procedure Baseline-Closure using a slight modification of Rytter's (1983, 1985) speedup for loop-free 2-PDAs. Let V be the set of all nodes of G (we have $|V| = n$), $p = \lceil \log n/2 \rceil$, and $r = \lceil n/p \rceil$. We use fast set representations of sets of nodes $X \subseteq V$—each such set is represented as a sequence r words, each of length p. We will need to convert a list representation of X into a fast set representation as above. It is easy to see that this can be done using a sort in $O(n \log n)$ time.

Now recall that the bottleneck of the baseline algorithm is line 19 of the routine Visit, which costs $O(mn)$ over an entire run of the algorithm. Now we show how to speed up this line. First, let us implement Baseline-Closure such that entries of the table T are stored as fast sets, and the sets $Out(u)$ are represented as lists.

/* speeds up the operation
$T(u) \leftarrow \bigcup_{v \in Out(u)} T(rep(v))$ */
let $x_1, \ldots, x_r$ be the words in the fast set for $Out(u)$ in
SPEEDUP()
1 compute $\langle x_1, \ldots, x_r \rangle$
2 **for** $1 \leq i \leq r$
3 **do if** $x_i = \mathbf{0}$ **continue**
4   **if** $\mathit{Cache}(i, x_i) = \perp$
5   **then** $\mathit{Cache}(i, x_i) \leftarrow \cup_{v \in Set(i, x_i)} T(rep(v))$
6     $T(u) \leftarrow T(u) \cup \mathit{Cache}(i, x_i)$

Figure 6.7: The speedup routine

Now consider the procedure SPEEDUP in Fig. 6.7, which is a way to speed up computation of the recurrence $T(u) \leftarrow \bigcup_{v \in Out(u)} T(rep(v))$. The idea is cache the value $(\cup_{v \in X} T(rep(v)))$ exhaustively for all non-empty sets X that are sufficiently small, and use this cache to compute the value for larger sets $Out(u)$. This is done using a table *Cache* (of global scope) such that for each $1 \leq i \leq r$ and for each word $w \neq \mathbf{0}$ of length p, we have a table entry $\mathit{Cache}(i, w)$ containing either a subset of V, represented as a fast set, or a special "null" value $\perp$ (note that the pair (i, w) uniquely identifies a subset of V of size at most p—this set is denoted by $Set(i, w)$). Initially, every entry of *Cache* equals $\perp$.

Let us now use the Assign-Union operation for fast sets (see Section 6.1) to implement line 6 of SPEEDUP, and replace line 19 of VISIT by a call to SPEEDUP. To see that this leads to a speedup, note that *Cache* has at most $r.2^p = O(n^{3/2}/\log n)$ entries. Now, line 5 in SPEEDUP gets executed at most once for each cell in *Cache* during a complete run of CLOSURE—i.e., $O(r.2^p) = O(n^{3/2}/\log n)$ times. Each time it is executed, it costs $O(n)$ time (as $Set(i, x_i)$ is of size $O(\log n)$ and as union of two entries of T costs $O(n/\log n)$ time), so that its *total* cost is $O(n^{5/2}/\log n)$. Thus, the bottleneck is line 6. Let us compute the total number of times this line is executed during a run of closure. Since the total size of all the $Out(u)$'s during a run of BASELINE-CLOSURE is bounded by m, the emptiness test in line 3 ensures that line 6 is executed $O(m)$

times in total during a run of the closure algorithm (this is the tighter bound when the graph is sparse). The other obvious bound on the number of executions of this line is $O(r.n)$ (this captures the dense case). Each time it is executed, it costs time $O(r)$. Thus, the total complexity of the modified algorithm (let us call this algorithm Closure) is $O(\min\{m.r, r.n.r\})$—i.e., $O(\min\{mn/\log n, n^3/\log^2 n\})$.

As for the space requirement of the algorithm, each fast set stored in a cell of the table *Cache* costs space $O(n)$. As *Cache* has $O(n^{3/2}/\log n)$ cells, the total cost of maintaining this table is $O(n^{5/2}/\log n)$. The space costs of the other data structures, including the table needed for fast sets operations if unit-cost word operations are not available, is subsumed by this cost. Hence we have:

Theorem 22. Closure *computes the transitive closure of a directed graph with n nodes and m edges in*

$$O(\min\{mn/\log n, n^3/\log^2 n\})$$

time and $O(n^{5/2}/\log n)$ space.

6.3.2 Bounded-stack RSMs

Using the ideas discussed earlier in this section, the algorithm Closure can now be massaged into a reachability algorithm for bounded-stack RSMs. Figure 6.8 shows pseudocode for a baseline algorithm for same-context reachability in bounded-stack RSMs obtained by modifying Baseline-Closure. The sets $H^s(u)$ in the new algorithm correspond to the sets *Reach*(u) in the transitive closure algorithm. The main difference lies in lines 14–17, which insert the summary edges into the graph. Also, as it is same-context reachability that we are computing, a child is added to the set *Out*(u) only if it is reached along a local or summary edge (the "else" condition in line 17). A correctness argument may be given following the discussion earlier in this section.

Adding an extra transitive closure step at the end of this algorithm gives us an algorithm for reachability. With some extra book-keeping, it is possible to evade this last step and compute reachability and same-context reachability in the same search—we omit the details. The speedups discussed earlier in this section may now be applied. Let us call the resultant algorithm Stack-Bounded-Reachability. It is easy to see that its complexity is the same as that of Closure. The only extra overhead is that of inserting the summary edges, and it is subsumed by the costs of the rest of the algorithm. Thus, the algorithm Stack-Bounded-Reachability has time complexity $O(\min\{mn/\log n, n^3/\log^2 n\})$, where m and n are the number of edges and nodes in the summary graph of the RSM. The space complexity is as for Closure. In general, m is $O(n^2)$, so that:

Theorem 23. *The algorithm* Stack-Bounded-Reachability *computes all-pairs reachability in a bounded-stack RSM of size n in $O(n^3/\log^2 n)$ time and $O(n^{5/2}/\log n)$ space.*

We note that an algorithm as above cannot be obtained from any of the existing sub-cubic algorithms for graph transitive closure. All previously known $O(n^3/\log^2 n)$-time algorithms for graph transitive closure use reductions to boolean matrix multiplication and do not permit online edge addition even if, as is the case for bounded-stack RSMs, these edges arise in a special way. While Chan [Cha05] has observed that DFS-based transitive closure may be computed in time $O(mn/\log n)$ using fast sets, this complexity does not suffice for our purposes.

6.4 Reachability in hierarchical state machines

As we saw, the reason why reachability in bounded-stack RSMs is easier than general RSM-reachability is that summary edges in the former case have a "depth-first" structure. For hierarchical state machines, the structure of summary edges is restricted enough to permit an algorithm with the same complexity as boolean matrix multiplication.

Let us have as input a hierarchical state machine M with components $M_1, \ldots, M_k$, such that a call from the component M_i can only lead to a component M_j for $j > i$. The summary graph H of M may be partitioned into k subgraphs $H_1, \ldots, H_k$ such that call-edges only run from partitions H_i to partitions H_j, where $j > i$. As the component M_k does not call any other component, there are no summary edges in H_k.

To compute reachability in M, first compute the transitive closure of H_k. Next, for all entries en and exits ex of M_k and all boxes b with $Y(b) = k$, add summary edges $((b, en), (b, ex))$. Now remove the call edges from H_{k-1} and compute its transitive closure and, once this is done, use the newly discovered reachability relations to create new summary edges in subgraphs H_j, where $j < k - 1$. Note that we do not need to process the graph H_k again. We proceed inductively, processing every H_i only once. Once the transitive closure of H_1 is computed, we add all the call edges from the different H_1's and compute the transitive closure of the entire graph. By Lemma 8, there is an edge from v to v' in the final closure iff v' is reachable from v.

As for complexity, let n be the total number of states in $\mathcal{A}$, and let n_i be the number of states in the subgraph H_i. Let $BM(n) = O(n^{2.376})$ be the time taken to multiply two $n \times n$ boolean matrices. Since transitive closure of a finite relation may be reduced to boolean matrix multiplication, the total cost due to transitive closure computation in the successive phases, as well as the final transitive closure, is $\Sigma_i BM(n_i) + BM(n) = O(BM(n))$. The total cost involved in identifying and inserting the summary and call edges is $O(n^2)$. Assuming $BM(n) = \omega(n^2)$, we have:

Theorem 24. *All-pairs reachability in hierarchical state machines can be solved in time $O(BM(n))$, where $BM(n) = O(n^{2.376})$ is the time taken to multiply two $n \times n$ boolean matrices.*

Of course, the above procedure is far from compelling—the cubic, summarization-based reachability algorithm published in the original reference on the analysis of these machines [AY98] is going to outperform it in any reasonable application. However, taken together with our other results, it highlights a gradation in the structure of the summary graph and the complexity of RSM-reachability as recursion in the input RSM is constrained.

```
Visit(u)
  add u to Visited
  push(u, L)
  low(u) ← dfsnum(u) ← height(L)
  H^s(u) ← ∅;    rep(u) ← ⊥
  Out(u) ← ∅
  if u is an internal state
    then Next(u) ← {v : u → v}
    else if u is a call (b, en)
           then Next(u) ← {en}
           else Next(u) ← ∅
  for v ∈ Next(u)
  do if v ∉ Visited then Visit (v)
     if v ∈ Done
       then if u = (b, en) is a call and v = en
              then for exit states ex ∈ H^s(en)
                     do add (b, ex) to Next(u)
              else add v to Out(u)
       else low(u) ← min(low(u), low(v))
  if low(u) = dfsnum(u)
    then repeat
              v ← pop(L)
              add v to Done
              add v to H^s(u)
              Out(u) ← Out(u) ∪ Out(v)
              rep(v) ← u
         until v = u
         H^s(u) ← H^s(u) ∪ ⋃_{v∈Out(u)} H^s(rep(v))
```

```
Baseline-Same-Context-Stack-Bounded-Reachability()
  Visited ← ∅;    Done ← ∅
  for each state u
  do if u ∉ Visited then Visit (u)
```

Figure 6.8: Same-context reachability in bounded-stack RSMs

Chapter 7

Conclusion

In this thesis, we have covered two aspects of software model checking: requirement specification and algorithmic analysis. It is accepted wisdom in software model checking and program analysis that any reasonably precise analysis of procedural programs needs to take into account their context-sensitive control flow; insofar, context-sensitive abstractions have received a lot of attention in recent times. In this thesis, we have argued that:

1. Temporal specification logics like the μ-calculus, while mainstays of traditional model checking, cannot specify context-sensitive program requirements such as: "A file is read before control leaves the current procedural context." The difficulty is that the mu-calculus is based on regular tree languages, which are not expressive enough to capture nesting of contexts. We have shown that a way to overcome this issue is to define temporal specifications based on regular languages of nested trees, which are a new class of graphs that we introduce.

2. While *context-sensitive reachability analysis* was for long believed to be intrinsically cubic, an asymptotically subcubic algorithm is possible.

In the former direction, we have re-phrased the branching time model checking question for context-sensitive abstractions. Such an abstraction is now viewed as generating a nested tree on which specifications are interpreted. The model checking question becomes: does this nested tree satisfy an NT-μ property? As for concrete specification notations, we have offered a fixpoint logic —called NT-μ— for nested trees. While this logic can express context-sensitive program requirements such as pre/post-conditions, combinations of local and global temporal properties, and stack-sensitive security requirements, it admits tractable, symbolic model checking. The logic unifies and generalizes many existing logics and fixpoint computations, identifying a new class of decidable properties of programs, and also allows modular, composable specifications. We have also introduced automata on nested trees defining *regular languages of nested trees*. We have shown that these languages have many of the attractive properties of tree automata— for example, they are closed under logical operations and allow model checking— while being far more expressive— for example, they can capture context-sensitive requirements and have an undecidable language emptiness problem. In a result that "lifts" the equivalence between the μ-calculus and alternating parity tree automata and suggests that NT-μ is a canonical calculus for these structures, we have shown NT-μ and alternating parity automata on nested trees are interconvertible. We have also explored monadic second-order logic in this setting, even if we have not found it to be too robust.

We believe that this theory of context-sensitive specifications will be a basis for future software model checking tools. As a first step in this more applied direction, we have designed a software specification language—called PAL— that can express context-sensitive safety requirements, but can yet be easily integrated into existing software analysis frameworks. To demonstrate this, we have implemented PAL on top of the software model checker BLAST [HJM+02].

In the second direction, we have given a method to perform reachability analysis of context-sensitive program abstractions or pushdown systems using efficient set operations. The algorithms reformulate a scheme called summarization that identifies "summary edges" capturing computations within the same context. However, while previous algorithms following this scheme were cubic, our algorithms achieve subcubic complexity, improving a long-known upper bound. We have also identified a gradation in the difficulty of reachability computation as recursion in the abstraction is restricted. We have observed that in reachability analysis of general recursive state machines, summary edges can arise in arbitrary orders, and all-pairs reachability can be determined in time $O(n^3/\log n)$. For bounded-stack RSMs, summary edges have a "depth-first" structure, and the problem can be solved in $O(n^3/\log^2 n)$ time using a modification of a DFS-based transitive closure algorithm. For hierarchical state machines, where there is no recursion, the states can be partitioned such that one needs to compute the closure of each partition only once.

7.0.1 Future work

Regarding future directions and open questions, there are many. So far as context-sensitive specifications are concerned, we note that our decision procedure for model checking NT-μ is very different from known methods for branching-time model-checking of context-sensitive abstractions [Wal01, BS99]. The latter seem too complex to work in practice; our algorithm, being symbolic in nature, appears more implementable. Also, note that our algorithm directly implements the operational semantics of NT-μ formulas over bounded summaries. In fact, in this regard NT-μ resembles the modal μ-calculus in the setting of finite-state systems whose formulas encode fixpoint computations over sets; to model-check μ-calculus formulas, we merely need to perform these computations. Unsurprisingly, our procedure is very similar to classical symbolic model-checking for the μ-calculus. There is one key difference, however: in the latter setting, efficient data structures such as binary decision diagrams (BDDs) are used to represent sets of system states. On the other hand, we do not yet know of an efficient data structure to store sets of bounded summaries. Note that the problem is more complex than that of representing sets of states; in our case, we need to store sets of tuples of the form $\langle v, U_1, \ldots, U_k \rangle$, where v is a state and the V_i-s are sets of states. We leave this as an open question.

There are a number of other questions of a more theoretical flavor. Note that we have left open the problem of establishing that MSO-logic on nested trees cannot capture the third-order fixpoints of NT-μ. Also note that our MSO-logic is undecidable and not very robust. Does it, however, have more attractive fragments? There are questions about NT-μ: what would, for instance, be the right axiomatization of this logic? Also, note that NT-μ expresses properties using *forward* modalities. As argued in [Sch98], several dataflow analysis problems also require backward modalities; extending NT-μ to backward modalities will result in expressing several other dataflow problems. How would this affect expressiveness and decidability? In general, we believe that nested trees are conceptually fundamental and merit further study. It would be interesting to examine languages of nested trees in other ways: do they, for example, have algebraic characterizations?

As for applications, there is the challenge of convincing the designers of popular program analysis and software model checking environments to use a language like Pal for expressing context-sensitive program requirements. We have been exploring possibilities of integrating the Pal specification language into an official release of the Blast model checker. Also, we believe that applications of nested trees beyond program verification are possible. Nested word structures are already known to have connections with XML query languages, since XML documents have a natural matching tag structure that can be modeled by jump-edges. Do nested trees have similar applications?

In the second direction, given that RSM-reachability is a central algorithmic problem in program analysis, the natural next step is to evaluate the practical benefits of these contributions. Such an effort should remember that real implementations of RSM-reachability-based program analyses apply heuristics such as cycle elimination and node clustering, and are often fine-tuned to the specific problem at hand. Thus, instead of implementing our algorithms literally, the goal should be to explore combinations of techniques known to work in practice with the high-level ideas used in this paper.

As for algorithmic directions, a natural question is whether this is the best we can do. A hard open question is whether all-pairs CFL-reachability can be reduced to boolean matrix multiplication. This would be especially satisfactory as the former can be trivially seen to be as hard as the latter. Yannakakis [Yan90] has noted that Valiant's reduction of context-free recognition to boolean matrix multiplication [Val75] can be applied directly to reduce CFL-reachability in *acyclic* graphs to boolean matrix multiplication. However, there seem to be basic difficulties in extending this method to general graphs.

Another set of questions involves stack-bounded RSMs and our transitive closure. Given a program without infinite recursion, can we automatically generate a stack-bounded abstraction that can be analyzed faster than a general RSM abstraction? Can our transitive closure algorithm have applications in other areas—for example, databases? Recall that, being a search-based algorithm, it does not require the input graph to be explicitly represented, and is suitable for computing partial closure—i.e., computing the sets of nodes reachable from some, rather than all, nodes. Algorithms with such features have been studied with theoretical as well as practical motivations— a new engineering question would be to see how well the techniques of this paper combine with them.

Bibliography

[ABE$^+$05] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. Reps, and M. Yannakakis. Analysis of recursive state machines. *ACM Transactions on Programming Languages and Systems*, 27(4):786–818, 2005.

[ACM06a] R. Alur, S. Chaudhuri, and P. Madhusudan. A fixpoint calculus for local and global program flows. In *Proceedings of the 33rd Annual ACM Symposium on Principles of Programming Languages*, 2006.

[ACM06b] R. Alur, S. Chaudhuri, and P. Madhusudan. Languages of nested trees. In *Computer-Aided Verification, CAV'06*, 2006.

[ADKz70] V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradžev. On economical construction of the transitive closure of an oriented graph. *Soviet Mathematics Doklady*, 11:1209–1210, 1970.

[AEM04] R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *TACAS'04: Tenth International Conference on Tools and Algorithms for the Construction and Analysis of Software*, LNCS 2988, pages 467–481. Springer, 2004.

[AF03] M. Abadi and C. Fournet. Access control based on execution history. In *NDSS*, 2003.

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Series in Computer Science and Information Processing. Addison-Wesley, 1974.

[AM04] R. Alur and P. Madhusudan. Visibly pushdown languages. In *Proceedings of the 36th ACM Symposium on Theory of Computing*, pages 202–211, 2004.

[AM06] R. Alur and P. Madhusudan. Adding nesting structure to words. In *Developments in Language Theory*, 2006.

[AY98] R. Alur and M. Yannakakis. Model checking of hierarchical state machines. In *Proceedings of the Sixth ACM Symposium on Foundations of Software Engineering*, pages 175–188. 1998.

[BBFG04] F. Besson, T. Blanc, C. Fournet, and A. D. Gordon. From stack inspection to access control: A security analysis for libraries. In *CSFW*, pages 61–75, 2004.

[BCC+03] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G.T. Leavens, R. Leino, and E. Poll. An overview of JML tools and applications. In *Proceedings of the 8th International Workshop on Formal Methods for Industrial Critical Systems*, pages 75–89, 2003.

[BCD+92] J.R. Burch, E.M. Clarke, D.L. Dill, L.J. Hwang, and K.L. McMillan. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[BCH+04] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. The BLAST query language for software verification. In *SAS*, pages 2–18, 2004.

[BCLR04] T. Ball, B. Cook, V. Levin, and S. K. Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In *IFM*, pages 1–20, 2004.

[BEM97] A. Boujjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Applications to model checking. In *CONCUR'97: Concurrency Theory, Eighth International Conference*, LNCS 1243, pages 135–150. Springer, 1997.

[BMMR01] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of c programs. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 203–213, New York, NY, USA, 2001. ACM Press.

[BR00] T. Ball and S. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN 2000 Workshop on Model Checking of Software*, LNCS 1885, pages 113–130. Springer, 2000.

[BR01] T. Ball and S. Rajamani. The SLAM toolkit. In *Computer Aided Verification, 13th International Conference*, 2001.

[BS99] O. Burkart and B. Steffen. Model checking the full modal mu-calculus for infinite sequential processes. *Theoretical Computer Science*, 221:251–270, 1999.

[Büc62] J.R. Büchi. On a decision method in restricted second-order arithmetic. In *Proceedings of the International Congress on Logic, Methodology, and Philosophy of Science 1960*, pages 1–12. Stanford University Press, 1962.

[CA07] S. Chaudhuri and R. Alur. Instrumenting c programs with nested word monitors. In *SPIN*, 2007.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[CDG+02] H. Comon, M. Dauchet, R. Gilleron, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Draft, Available at `http://www.grappa.univ-lille3.fr/tata/`, 2002.

[CE81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logic of Programs*, LNCS 131, pages 52–71. Springer-Verlag, 1981.

[CGJ+00] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.

[CGP99] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999. CLA e 99:1 1.Ex.

[Cha05] T. M. Chan. All-pairs shortest paths with real weights in $o(n^3/\log n))$ time. In *WADS*, pages 318–324, 2005.

[Cha07] T. M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. In *STOC*, pages 590–598, 2007.

[CMM+04] K. Chatterjee, D. Ma, R. Majumdar, T. Zhao, T.A. Henzinger, and J. Palsberg. Stack size analysis for interrupt driven programs. *Information and Computation*, 194(2):144–174, 2004.

[CW02] H. Chen and D. Wagner. Mops: an infrastructure for examining security properties of software. In *Proceedings of ACM Conference on Computer and Communications Security*, pages 235–244, 2002.

[EJ91] E.A. Emerson and C.S. Jutla. Tree automata, mu-calculus, and determinacy. In *Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science*, pages 368–377, 1991.

[EKS77] J. Eve and R. Kurki-Suonio. On computing the transitive closure of a relation. *Acta Inf.*, 8:303–314, 1977.

[EKS03] J. Esparza, A. Kucera, and S. S. Schwoon. Model-checking LTL with regular valuations for pushdown systems. *Information and Computation*, 186(2):355–376, 2003.

[Eme90] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–1072. Elsevier Science Publishers, 1990.

[Flo67] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.

[GTW02] E. Grädel, W. Thomas, and T. Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*, volume 2500 of *Lecture Notes in Computer Science*. Springer, 2002.

[HJM+02] T.A. Henzinger, R. Jhala, R. Majumdar, G.C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *CAV 02: Proc. of 14th Conf. on Computer Aided Verification*, LNCS 2404, pages 526–538. Springer, 2002.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hol91] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[Hol97] G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[HRB88] S. Horwitz, T. W. Reps, and D. Binkley. Interprocedural slicing using dependence graphs (with retrospective). In *Best of PLDI*, pages 229–243, 1988.

[HRS95] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *SIGSOFT '95: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, pages 104–115, New York, NY, USA, 1995. ACM Press.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[JMT99] T. Jensen, D. Le Metayer, and T. Thorn. Verification of control flow based security properties. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 89–103, 1999.

[JW96] D. Janin and I. Walukiewicz. On the expressive completeness of the propositional mu-calculus with respect to monadic second order logic. In *CONCUR'96: Seventh International Conference on Concurrency Theory*, LNCS 1119, pages 263–277. Springer-Verlag, 1996.

[KMM00] M. Kaufmann, J. S. Moore, and P. Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[Koz83] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[KPV02] O. Kupferman, N. Piterman, and M.Y. Vardi. Pushdown specifications. In *Proc. 9th Intl. Conf. on Logics for Programming, Artifical Intelligence, and Reasoning*, LNCS 2514, pages 262–277. Springer, 2002.

[LÖ5] C. Löding. Private communication. 2005.

[McM93] K.L. McMillan. *Symbolic model checking: an approach to the state explosion problem.* Kluwer Academic Publishers, 1993.

[MR00] D. Melski and T. W. Reps. Interconvertibility of a class of set constraints and context-free-language reachability. *Theor. Comput. Sci.*, 248(1-2):29–98, 2000.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[NST01] N. Nitta, H. Seki, and Y. Takata. Security verification of programs with stack inspection. In *SACMAT '01: Proceedings of the sixth ACM symposium on Access control models and technologies*, pages 31–40, New York, NY, USA, 2001. ACM Press.

[ORR+96] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, LNCS 1102, pages 411–414. Springer-Verlag, 1996.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–77, 1977.

[Pur70] P. W. Purdom. A transitive closure algorithm. *BIT*, 10:76–94, 1970.

[QS82] J.P. Queille and J. Sifakis. Specification and verification of concurrent programs in CESAR. In *Proceedings of the Fifth International Symposium on Programming*, LNCS 137, pages 195–220. Springer-Verlag, 1982.

[Rab69] M.O. Rabin. Decidability of second order theories and automata on infinite trees. *Transactions of the AMS*, 141:1–35, 1969.

[Rep95] T. Reps. Shape analysis as a generalized path problem. In *PEPM '95: Proceedings of the 1995 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 1–11, New York, NY, USA, 1995. ACM Press.

[Rep98] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701–726, 1998.

[RF01] J. Rehof and M. Fähndrich. Type-base flow analysis: from polymorphic subtyping to cfl-reachability. In *POPL*, pages 54–66, 2001.

[RHS95] T. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 49–61, 1995.

[Rou70] William C. Rounds. Mappings and grammars on trees. *Mathematical Systems Theory*, 4(3):257–287, 1970.

[Ryt83] W. Rytter. Time complexity of loop-free two-way pushdown automata. *Inf. Process. Lett.*, 16(3):127–129, 1983.

[Ryt85] W. Rytter. Fast recognition of pushdown automaton and context-free languages. *Information and Control*, 67(1-3):12–22, 1985.

[Sch83] L. Schmitz. An improved transitive closure algorithm. *Computing*, 30:359–371, 1983.

[Sch98] D.A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages*, pages 68–78, 1998.

[SGSB05] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for java. In *OOPSLA*, pages 59–76, 2005.

[SH97] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proc. 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 72–83. Springer Verlag, 1997.

[SP81] M. Sharir and A. Pnueli. Two approaches to interprocedural dataflow analysis. *Program Flow Analysis: Theory and Applications*, pages 189–234, 1981.

[Ste91] B. Steffen. Data flow analysis as model checking. In *Theoretical Aspects of Computer Software: TACS'91*, LNCS 526, pages 346–365, 1991.

[Sti91] C.S. Stirling. Modal and temporal logic. In *Handbook of Logic in Computer Science*, pages 477–563. Oxford University Press, 1991.

[Tho97] W. Thomas. Languages, automata, and logic. *Handbook of Formal Language Theory*, III:389–455, 1997.

[Val75] L. G. Valiant. General context-free recognition in less than cubic time. *J. Comput. Syst. Sci.*, 10(2):308–315, 1975.

[Wal01] I. Walukiewicz. Pushdown processes: Games and model-checking. *Information and Computation*, 164(2):234–263, 2001.

[WF98] Dan S. Wallach and Edward W. Felten. Understanding Java stack inspection. In *IEEE Symp. on Security and Privacy*, pages 52–63, 1998.

[Yan90] M. Yannakakis. Graph-theoretic methods in database theory. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, pages 230–242, 1990.