

Tel-Aviv University
Raymond and Beverly Sackler Faculty of Exact Sciences
School of Computer Science

**EMPLOYING DECISION PROCEDURES
FOR VERIFICATION OF HEAP-MANIPULATING PROGRAMS**

by

Greta Yorsh

under the supervision of
Prof. Mooly Sagiv and Prof. Alexander Rabinovich

Thesis submitted
for the degree of Doctor of Philosophy

Submitted to the Senate of Tel-Aviv University
December 2007

Dedicated to the memory of my grandmother, Berta Katz.

Abstract

Employing Decision Procedures
for Verification of Heap-Manipulating Programs

Greta Yorsh
Doctor of Philosophy
School of Computer Science
Tel-Aviv University

The goal of software verification is to guarantee the reliability of software via rigorous methods that can establish its correctness, or to detect subtle design errors. As the size and the complexity of software grows, verification tasks become more challenging.

The first part of this thesis provides novel algorithms that harness automated reasoning tools (e.g., theorem provers and decision procedures) to perform program analysis and verification. These algorithms automate the process of developing program analyses, for instance, by computing the precise effect of program statements.

While these algorithms are applicable to a wide range of analysis problems, the main focus of this thesis is analysis of programs that manipulate linked data-structures, such as singly-linked lists, doubly-linked lists, trees, etc. Specifications of these programs often involve properties regarding reachability (via pointer dereference) between heap-allocated objects, e.g., to establish that a data-structure is acyclic; every element is reachable from the root of the data-structure; two data-structures are disjoint.

The second part of this thesis provides a way to automatically reason about interesting reachability properties, using a new decidable logic, *LRP*. A decision procedure for *LRP* can be employed in the algorithms developed in the first part of this thesis.

Acknowledgements

I have been very fortunate to have Mooly Sagiv and Alex Rabinovich as my advisors.

I thank Alex for teaching me what it takes to write elegant proofs: to grasp the deep semantic meaning of problems and to distill their essence.

Mooly’s scientific knowledge, creativity, and devotion to research define the meaning of Scientist. I aspire to it. At the finish line of this “marathon”, having already registered for the next one, I would like to thank Mooly for being my coach throughout the course. I cannot thank him enough for his dedicated guidance.

Most of the research presented in this thesis is joint work with others: Mooly Sagiv, Alex Rabinovich, Tom Reps, Tom Ball, Antoine Meyer, and Ahmed Bouajjani [YRS04, YBS06, YBS07, YRS+06, YRS+07].

I am indebted to Tom Reps for his tremendous influence on me as a researcher from the very beginning. I have benefitted greatly from discussions with Tom and his students, during my two short visits to the University of Wisconsin, Madison.

I thank Tom Ball for introducing me to many cool problems, for providing wise advice, and for encouraging me to believe in myself. I work on my juggling skills too.

During my graduate studies, I had the extraordinary opportunity to work with Reinhard Wilhelm, Neil Immerman, and Orna Kupferman. I learned so much from every single discussion with each one of them.

I had fruitful internships in industrial research labs, thanks to collaboration with Madan Musuvathi (Microsoft Research Redmond), Satish Chandra (IBM Research), and Byron Cook (Microsoft Research Cambridge).

Mooly’s research group is unique in its excellence. It has been extremely inspiring to work in this group. I thank my “academic siblings”, Nurit Dor, Tal Lev-Ami, Roman Manevich, Noam Rinetzky, and Eran Yahav, for their frequent feedback on my ideas, their valuable comments, insights and for coping with my moods at times.

It has always been a joy discussing ideas with colleagues who are also friends: Joerg Bauer, Sumit Gulwani, Mayur Naik, Mike Weber, and Viktor Kuncak. (Scarcity of women in our field strikes again.)

I was only able to complete the writing of my dissertation thanks to the support and inspiration from Josh Berdine.

I would like to thank the Israeli Academy of Science and the Levi-Eshkol fellowship for their generous financial support throughout my studies.

I am intensely grateful to all my Iras and other good friends for their moral support. Special thanks to Rina Talisman, for “kicking my ass” for so many years.

Finally and most importantly, I thank my dear mother, Irina Gregorievna Yorsh, for her constant belief in me. I am very happy that I keep living up to it.

Contents

1	Introduction	1
1.1	Thesis Contributions	1
1.2	Thesis Organization	2
1.3	Overview	2
1.3.1	Combining Concrete Execution, Abstraction and Theorem Proving	3
1.3.2	Symbolically Computing Most-Precise Abstract Operations for Shape Analysis	5
1.3.3	Comparison between the Algorithms	8
1.3.4	The Role of a Theorem Prover	8
1.3.5	The Logic of Reachable Patterns	9
2	Combining Concrete Execution, Abstraction, and Theorem Proving	11
2.1	Introduction	11
2.2	Example	12
2.2.1	Finding a Bug	13
2.2.2	Finding a Proof	13
2.2.3	Finding a False Error	15
2.3	Formal Description	15
2.3.1	Abstraction and Concretization	15
2.3.2	Basic Procedure	16
2.3.3	Symbolic Procedure	18
2.4	Towards a Realistic Implementation	19
2.4.1	Program Analysis Infrastructure	19
2.4.2	Cutpoints	19
2.4.3	On-the-fly Abstraction	19
2.4.4	Interprocedural Analysis	20
2.4.5	Employing a Theorem Prover	20
2.4.6	Controlling Concrete Execution	20
2.4.7	Hybrid Approach	21
2.5	Prototype Implementations	21
2.5.1	Based on Predicate Abstraction and XRT	21
2.5.2	Based on Canonical Abstraction and TVLA	24
2.6	Avoiding Unnecessary Abstraction Refinement	24
2.7	Related Work	25
3	Computing Most-Precise Abstract Operations for Shape Analysis	28
3.1	Overview of Canonical Abstraction	29
3.1.1	3-Valued Structures	29
3.1.2	Embedding Order on 3-Valued Structures	31
3.1.3	Integrity Rules	31
3.1.4	Canonical Abstraction	31
3.2	The <i>assume</i> Algorithm	32
3.2.1	Employing a Theorem Prover	33

3.2.2	Materialization	34
3.2.3	Refining Relation Values	35
3.2.4	Properties of the Algorithm	35
3.2.5	Computing $\hat{\alpha}$	36
3.3	Implementing the Best Transformer	36
3.4	Related Work	36
4	Logic of Reachable Patterns in Linked Data-Structures	38
4.1	The \mathcal{L}_0 Logic	38
4.1.1	Syntax of \mathcal{L}_0	38
4.1.2	Semantics of \mathcal{L}_0	40
4.1.3	Finite Model Property	41
4.2	Undecidability of \mathcal{L}_0	42
4.3	Decidable and Useful Fragment of \mathcal{L}_0	43
4.3.1	The \mathcal{L}_1 Fragment	43
4.3.2	Describing Linked Data-Structures in \mathcal{L}_1	44
4.3.3	Expressing Verification Conditions in \mathcal{L}_1	45
4.3.4	Characterizing Shape Abstractions in \mathcal{L}_1	47
4.4	Decidability of \mathcal{L}_1	49
4.4.1	Translation from \mathcal{L}_0 to MSO	49
4.4.2	Decidability of MSO on Ayah Graphs	50
4.4.3	Normal Form of \mathcal{L}_0 Formulas	53
4.4.4	Decidability of \mathcal{L}_1	54
4.5	Ayah Model Property of \mathcal{L}_1	54
4.5.1	Trees with Extra Edges	55
4.5.2	Ayah Graphs	57
4.5.3	Graph Operations Enabled by \mathcal{L}_1 Formulas	57
4.5.4	Homomorphism Preservation	58
4.5.5	Witness Splitting	59
4.5.6	\mathcal{A}_k -Model Property of \mathcal{L}_1	61
4.6	The \mathcal{L}_2 Fragment and its Decidability	62
4.6.1	\mathcal{A}_k^{rem} -Model Property of \mathcal{L}_2	63
4.6.2	MSO is decidable on \mathcal{A}_k^{rem}	63
4.7	Complexity	64
4.7.1	Lower Bound: \mathcal{L}_1 is NEXPTIME-hard	65
4.7.2	Upper Bound: \mathcal{L}_1 is in 2EXPTIME	67
4.8	Limitations and Further Extensions	69
4.8.1	The Logic \mathcal{L}_3	69
4.8.2	The Logic $U\mathcal{L}_1$	70
4.9	Related Work	70
5	Conclusions and Future Work	72
	Bibliography	74
A	Appendix for Chapter 2	80
A.1	Lattice operations	80
A.2	Proofs	80
B	Proofs for Chapter 3	82

List of Figures

1.1	Combining Concrete Execution, Abstraction and Theorem Proving	3
1.2	Schematic view of the algorithms.	5
1.3	Symbolic algorithms for most-precise abstract operations for shape analysis.	7
2.1	The procedure <code>foo</code> contains a null pointer dereference error at line G.	12
2.2	Reachable abstract states <code>foo</code> using different abstraction functions.	14
2.3	The basic procedure.	16
2.4	Example program and abstract state space.	21
2.5	Implementation of a bounded stack using fixed-size array.	23
2.6	Analysis results for methods that manipulate singly-linked lists.	24
2.7	Reachable abstract states of two-process Bakery protocol, using an abstraction function $x_1 \leq x_2$	25
3.1	Relations for <code>List</code> data-type	29
3.2	A declaration of a linked-list data-type in C.	30
3.3	An abstract value a and the result of <code>assume[p](a)</code>	30
3.4	Concrete states represented by the structure S_1 from Fig. 3.3.	32
3.5	The <code>assume</code> algorithm.	33
3.6	The <code>bif</code> procedure.	34
3.7	A computation tree for <code>assume[p](a)</code> for a shown in Fig. 3.3.	35
4.1	A sketch of a grid model for a tiling problem.	42
4.2	Definitions of some useful patterns for \mathcal{L}_1	44
4.3	Properties of data-structures expressed in \mathcal{L}_1	45
4.4	The <code>reverse</code> procedure performs in-place reversal of a singly-linked list	46
4.5	An example graph that satisfies the VC_{loop} formula for <code>reverse</code>	46
4.6	The <code>append</code> procedure concatenates two singly-linked lists.	47
4.7	Examples of graphs in \mathcal{A}_0 , \mathcal{A}_1 , and \mathcal{A}_2	51
4.8	Merge operation on T^k -graphs.	56
4.9	Example of the construction used in the proof of Theorem 4.4.14.	59
4.10	Construction and homomorphisms in the proof of decidability.	62
4.11	The graph G_4	63

Chapter 1

Introduction

Software technologies affect a wide range of areas in today's world, starting from the way people communicate and interact with each other, and including safety-critical applications, such as aerospace and medicine technologies. As our dependence on software grows, the importance of software reliability increases. The goal of software verification is to guarantee the reliability of software via rigorous methods that can establish its correctness, or to detect subtle design errors. The verification process must take into account many complex and expressive features supported by modern programming languages. One of the main challenges is to handle unbounded resources such as dynamic data-structures with no given (or no reasonable) bound on their maximal size.

As the size and the complexity of software grows, it becomes more important to *automate* the verification tasks. Automatic software verification can be carried out by a static program analysis. It usually relies on *abstraction* to reason about all possible program executions, without actually executing the program. The choice of abstraction is guided by the program and the properties of interest. There is a trade-off between the precision of the abstraction and the cost (in terms of time and space) of the corresponding program analysis. Substantial progress has been made to develop scalable analyses that are sufficiently precise for certain classes of programs and properties. Currently, the focus is shifting to analysis of more complex programs and properties. In this setting, where significantly more precise and inherently more expensive analyses are required, scaling an analysis is no longer a matter of engineering; it becomes a research challenge that requires developing radically new approaches.

Theorem provers have been employed successfully to prove interesting properties of hardware and software systems, e.g., [MLK98, FLL⁺02]. In this thesis, we enrich the program analysis designer's toolbox with theorem provers: the designer of a program analysis defines an abstraction and the concrete meaning of basic statements, and our techniques harness theorem provers to automatically determine the abstract meaning of basic statements.

For parametric abstractions, our techniques alleviate the pain and suffering of manually computing the abstract meaning for every instance of the parametric abstraction. This is particularly important for expressive abstractions such as those used for verifying properties of heap-manipulating programs. Additionally, given a formal specification of a procedure, our techniques can be used to compute its abstract meaning. This enables modular reasoning in that the effect of calls to that procedure can be analyzed without using the code of the procedure. The abstract meaning computed in this way is guaranteed to be the most precise with respect to the given abstraction, under certain conditions, detailed later.

1.1 Thesis Contributions

The first part of this thesis (Chapters 2 and 3) provides two novel algorithms for computing an abstract representation of a set of concrete program states described by the specification. These algorithms allow program analysis tools to reason about human-provided specifications, and thus enable modular program analysis. For instance, we can use these algorithms to automatically compute the effect of a procedure call in any (abstract) calling context, using the procedure's formal specification. Moreover, these algorithms can be used to implement abstract transformers [CC79] and other abstract operations for parametric abstract domains, such as canonical abstraction [SRW02] and predicate abstraction [GS97]. Under certain conditions, the most precise result can be computed. These algorithms rely on automated reasoning tools (e.g., theorem provers and decision procedures).

While these algorithms are applicable to a wide range of analysis problems, the main focus of this thesis is analysis of programs that manipulate *linked data-structures*, such as singly-linked lists, doubly-linked lists, trees, etc.

Linked data-structures are important and widely used, in particular because they provide a way to efficiently handle an unbounded amount of data. However, this also makes it easy to introduce subtle errors, e.g., by violating global invariants, which cause a program to produce unexpected results and possibly crash. Therefore, it is important to verify programs that manipulate linked data-structures.

Specifications of these programs often involve *reachability* properties. For example, to establish that a memory configuration contains no garbage elements, we can show that every element is reachable from some program variable. Another example is acyclicity of data-structure fragments, i.e., every element reachable from node u cannot reach u .

For programs that manipulate linked data-structures, the success of the algorithms mentioned above depends on having a tool that supports automated reasoning about reachability. Automated reasoning about reachability is a difficult task. For instance, many reasoning problems which are decidable become undecidable when (even limited) support for reachability is added, e.g., [GME99, IRR⁺04a].

The second part of this thesis (Chapter 4) provides a way to automatically reason about interesting reachability properties, using a new *decidable* logic. This logic, called *LRP*, is both decidable and expressive enough to describe important properties of data-structures with an arbitrary number of pointer fields and of arbitrary shapes. A decision procedure for *LRP* can be employed, as a theorem prover, in the algorithms mentioned above. This allows us, for instance, to compute an abstract representation of the effect of calling a procedure, provided that its specification is expressed in *LRP*.

1.2 Thesis Organization

The main results of this thesis are described in (Chapters 2, 3, and 4), each of which can be read independently from others. In Section 1.3, we provide an informal overview of each of these chapters, and outline the connections between them. In Chapter 2, we present an algorithm that computes abstract representations of reachable program states using a novel combination of concrete execution, abstraction, and an automatic theorem prover. In addition, this algorithm explains the results of abstract interpretation in terms of concrete execution and abstraction, providing an intuitive introduction to the concepts that we use in the next chapter. Next, in Chapter 3, we present a different algorithm that is specialized for canonical abstraction, and thus, for reasoning about linked data-structures. When verifying properties of linked data-structures, both algorithms require an automated reasoning tool that can handle reachability properties. Motivated by this requirement, we develop a decidable logic that can express interesting reachability properties, as shown in Chapter 4. Finally, in Chapter 5, we summarize the results of this thesis and discuss future research directions.

1.3 Overview

This section provides an informal overview of the content of this thesis. The section contains forward references to chapters that formally discuss the presented material.

A Few Words on Terminology To provide some intuition to the readers who are not very familiar with abstract interpretation, the informal explanation in this section uses an abstract domain that is a powerset of “abstract states”. However, our algorithms are not restricted to powerset domains.

We use the term “most-precise abstract interpreter” for an abstract domain (with a finite height) to refer to the abstract interpreter that uses the best abstract transformers for all (intraprocedural) statements [CC79].

To simplify the presentation, we assume that an abstract value collectively describes states at all program points, rather than having a separate abstract value for each program point. This can be achieved by encoding the program counter in the representation of a concrete state.

Finally, we use the term “theorem prover” for an automatic tool for checking validity of formulas in a (decidable or undecidable) logic.

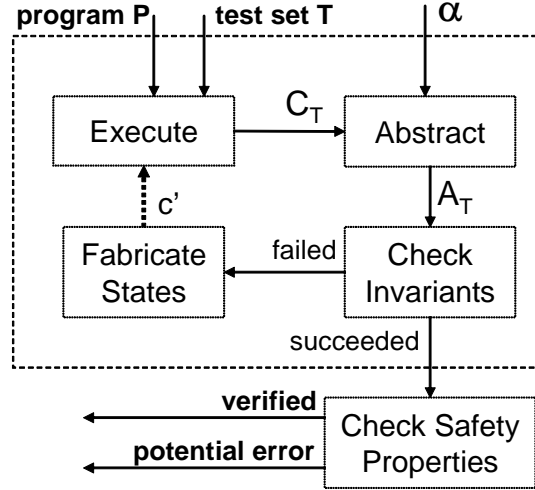


Figure 1.1: Overview of the method. C_T denotes the set of concrete states reachable from the states in T . A_T is the set of abstract states covered by T , i.e., $A_T = \alpha(C_T)$.

1.3.1 Combining Concrete Execution, Abstraction and Theorem Proving

It is well known that the problem of proving safety properties is undecidable in general. Fortunately, these properties often can be proved using abstraction to overapproximate the reachable concrete states of a program.

Abstraction and abstract interpretation [CC77] are key tools for automatically proving properties of systems, both for hardware [CGL94, Dam96] and software systems [NNH99]. An abstraction function α maps concrete program states to the corresponding abstract states. The concretization function γ maps every abstract state to the set of concrete states that it represents. A concrete state is reachable if it can arise in some program execution. A set of abstract states is *sound* if it represents all reachable concrete states of the program. An abstract state is reachable if it is the abstraction of some reachable concrete state. Identifying exactly the reachable abstract states is undecidable in general. Abstract interpretation provides a way to compute a *superset* of all reachable abstract states. Thus, the result of abstract interpretation can be used to check safety properties: if safety properties hold on (a superset of) all reachable abstract states, then these safety properties also hold on all reachable concrete states.

We propose a new method for computing a superset of all reachable abstract states. In contrast to abstract interpretation, which “executes” the program on abstract states, our method executes the program on concrete states, and then performs abstraction. Our method has five steps, shown in Fig. 1.1, as follows.

1. **Execute** Given a program P and a set T of test inputs,¹ execute the program and collect the concrete program states C_T obtained during execution.
2. **Abstract** Given an abstraction function α , obtain the set of abstract states: $A_T = \alpha(C_T)$. We say that A_T is the set of abstract states covered by T .
3. **Check Invariants** Check that A_T is invariant under the program P : if a concrete state is represented by A_T then its successor states are also represented by A_T . Formally, we check that for all concrete states c and c' such that $\alpha(\{c\}) \subseteq A_T$ and c' is a successor state of c in the program P , $\alpha(\{c'\}) \subseteq A_T$. This condition is expressed as a logical formula using strongest (liberal) postconditions [Dij76] such that if the formula is valid then A_T is an invariant.² The validity of the formula is checked using a theorem prover.
4. **Fabricate States** If A_T is not an invariant then there are concrete states c and c' such that $\alpha(\{c\}) \subseteq A_T$ and c' is a successor state of c in P , but $\alpha(\{c'\}) \not\subseteq A_T$. Our method finds such a state c' using a model generator, i.e., a theorem prover that produces a concrete counterexample for invalid formulas. We say that

¹Test input are represented as concrete states at the initial location of the program.

²Alternatively, a formula based on the weakest (liberal) precondition can be used, see Section 2.3.3 for details.

a state c' , obtained using a model generator as above, is a *fabricated state*. Note that a fabricated state is a concrete state at some intermediate program point, that is not necessarily reachable from any initial state of the program.

Then, our method augments T with c' and repeats the process, executing the program from a fabricated state, and so on, as shown in Fig. 1.1. This guarantees that the coverage increases in the next iteration, and if the process terminates, A_T is an invariant.

5. **Check Safety Properties** We check, using a theorem prover, whether the covered abstract states A_T satisfy the safety properties. If A_T is an invariant then it contains all reachable abstract states (assuming that the input test set T covers all initial program states). Thus, if A_T is an invariant, and the safety check succeeds on A_T , we have proven that all reachable concrete states of the program satisfy the safety properties. If the safety check fails, we report a *potential error*, which may indicate a real error in the program or a *false alarm*, due to the imprecision of the abstraction.

This algorithm allows us to perform modular program analysis using procedure specifications, as follows. When analyzing one module, in the “Execute” step, we can stop concrete execution at call sites of procedures, and proceed to the following steps of our algorithm (“Abstract” and “Check Invariants”). When checking invariants, we can use procedure specifications (instead of strongest postconditions), to compute the effect of a procedure call. This way, we can check invariants in a modular fashion, without using the code of the procedure. If the invariant check fails, it also provides a fabricated state at the program point where the procedure call returns to, and we can continue concrete execution from it.

Fig. 1.2(a) depicts the idea behind the iterative process in this algorithm. It shows the concrete and the abstract state spaces as the left and the right ovals, respectively. Each point in the right oval represents the value of A_T in some iteration of the algorithm, and the corresponding sets of concrete states are shown on the left.

Let X denote all reachable concrete states of the program (depicted in Fig. 1.2(a) as the dashed region inside the left oval). The goal of our algorithm is to compute $\alpha(X)$. This operation is not computable directly, because X can be infinite in general. In contrast, the “Abstract” step of our algorithm in each iteration computes $\alpha(C_T)$, where C_T is a finite set of concrete states. For most abstractions, the operation α can be easily computed for a finite set.

Our algorithm works its way up in the right oval, which on the left corresponds to progressively representing more and more concrete states, until the entire set X is represented. Of course, because of the inherent loss of information due to abstraction, the result can also represent concrete states outside of X . Under certain conditions, detailed later, the result of the algorithm represents the tightest set of concrete states that contains X and is expressible with the given abstraction.

The algorithm spends a significant amount of time in checking validity and model generation. Moreover, failure of these tasks (see Section 1.3.4) might cause loss of precision. An existing test set and a good choice of fabricated states allows us to reduce the number of iterations of the algorithm, and thus, the number of calls to a theorem prover and a model generator. Intuitively, in each iteration, we would like to “jump” further up in the right oval of Fig. 1.2(a), increasing the coverage as much as possible using abstraction of concrete states obtained from concrete executions, before performing the invariant check and the fabrication.

Properties of the Algorithm For finite-height abstract domains, our method is guaranteed to terminate and the result is the same as the result of the most-precise abstract interpreter (over the same abstract domain), assuming that all theorem prover calls were conclusive. In particular, our method produces the same false alarms as abstract interpretation.

It is noteworthy that we can make these guarantees even if we prematurely halt concrete execution in order to perform the coverage check. In this way, we can control the amount of time spent executing the program vs. the amount of time spent calling the theorem prover.

If the algorithm is stopped prematurely, before an invariant is found, then the results might be unsound (i.e., might miss errors) or imprecise (i.e., might produce false alarms). As opposed to this algorithm, in the algorithm described in Section 1.3.2, the intermediate results are always sound: they can be used for proving safety properties even if the analysis terminated prematurely (losing the ability to guarantee the precision), but that algorithm is less general.

As usual, our analysis (and abstract interpretation) does not distinguish between a false error and a real error. It is possible to combine our method (and abstract interpretation) with an analysis for classifying potential errors

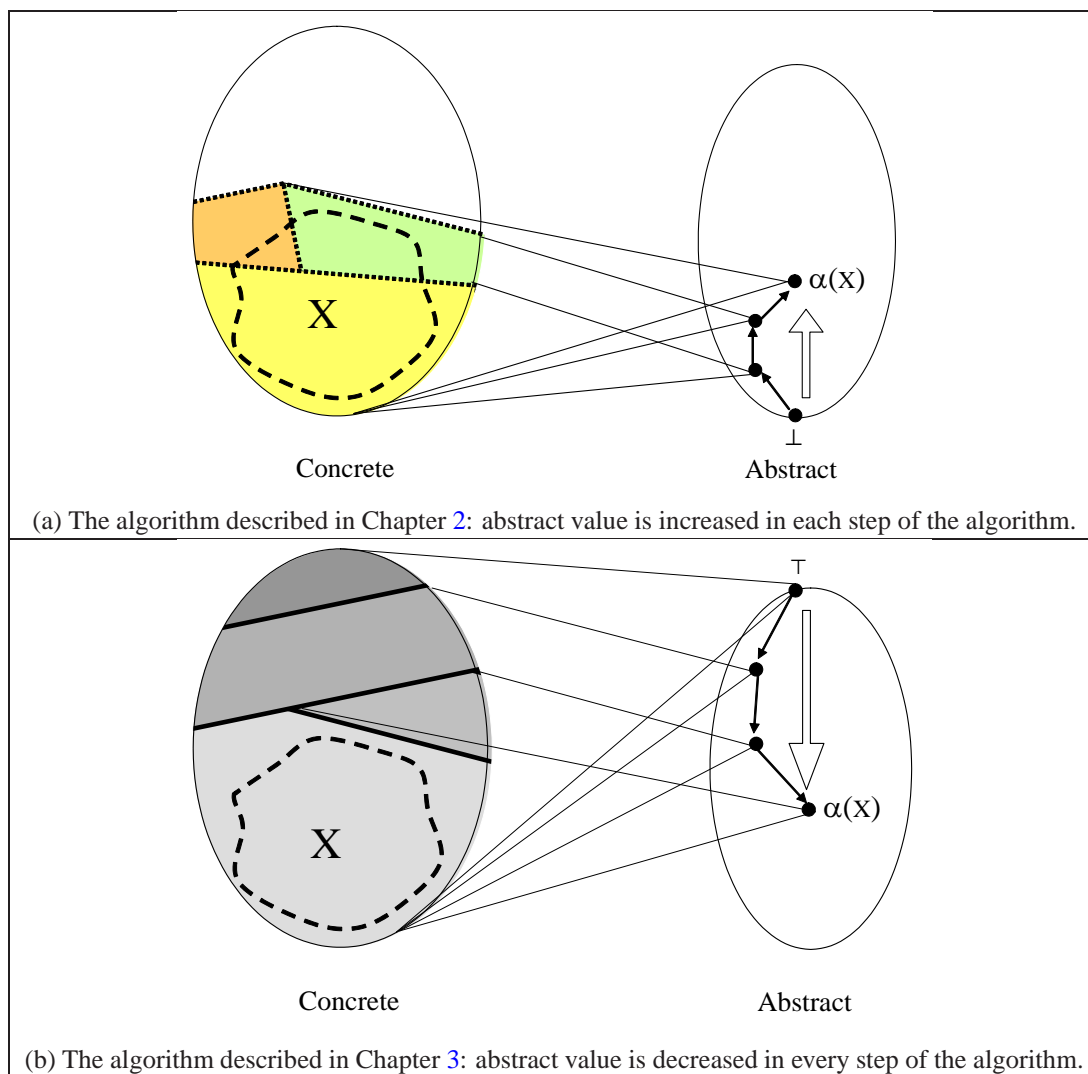


Figure 1.2: Schematic view of the algorithms.

into real errors and false alarms. In this thesis, we assume that the abstraction α is given. It is possible to combine our method with abstraction refinement to find a suitable abstraction.

Applications To evaluate the feasibility of this approach, we have implemented two prototypes: the first prototype uses predicate abstraction [GS97] and the XRT model checker [GTS05] infrastructure as its platform; the second prototype uses canonical abstraction [SRW02] and the TVLA system [LAS00] as its platform. The latter was used for checking memory safety properties of small but intricate programs that manipulate linked lists.

This material is described in detail in Chapter 2. It was originally published in [YBS06] and an extended version was invited for a journal publication in [YBS07]. This material is closely related to, and was inspired by, an earlier work [RSY04], as explained in Section 2.7.

1.3.2 Symbolically Computing Most-Precise Abstract Operations for Shape Analysis

The automatic verification of programs with dynamic memory allocation and pointer manipulation is a challenging problem. In fact, due to dynamic memory allocation and destructive updates of pointer-valued fields, the program

memory can be of arbitrary size and structure. This requires the ability to reason about a potentially unbounded number of memory structures, even for programming languages that have good capabilities for data abstraction. Usually abstract-datatype operations are implemented using loops, procedure calls, and sequences of low-level pointer manipulations; consequently, it is hard to prove that a data-structure invariant is reestablished once a sequence of operations is finished [Hoa75].

To tackle the verification problem of programs that manipulate dynamically allocated memory, several approaches emerged in the last few years with different expressive powers and levels of automation, including works based on abstract interpretation [LAS00, SRW02, RSW04, DOY06, LYY05], logic-based reasoning [IO01, Rey02], and automata-based techniques [KS93, MS01, BHPV05].

Shape-analysis algorithms based on canonical abstraction [SRW02] are capable of establishing that certain invariants hold for (imperative) programs that perform destructive updates on dynamically allocated storage. For example, they have been used to establish that a program preserves treeness properties, as well as that a program satisfies certain correctness criteria [LARSW00]. The TVLA system [LAS00] automatically constructs shape-analysis algorithms from a description of the operational semantics of a given programming language, and the shape abstraction to be used. The methodology of abstract interpretation has been used to show that the shape-analysis algorithms generated by TVLA are *sound* (conservative), but these algorithms do not necessarily compute the most-precise results with respect to the given abstraction.

To improve the precision of these algorithms, TVLA system uses Focus and Coerce operations. The Focus operation, also known as “partial concretization”, is a semantic reduction that is guided by user-specified formulas. Writing useful Focus formulas is not trivial — it may require understanding of the abstract domain and TVLA system. The Coerce operation uses Kleene evaluation to perform semantic reduction. Kleene evaluation can only recover very limited properties about the concrete states. The motivation of the work described below is to improve the precision, the scalability, and the automation of TVLA by employing a theorem prover.

Recall that the abstraction function α maps a potentially infinite set of concrete states to the (most-precise) abstract value for it. The concretization function γ maps an abstract value to the set of concrete states that the abstract value represents. In [Yor03, YRSW07], we introduce the symbolic operation $\hat{\gamma}$ which maps every abstract value a to a logical formula, called a *characteristic formula*, whose meaning is exactly the set $\gamma(a)$.³ That is, a concrete state is represented by a if and only if it satisfies the formula $\hat{\gamma}(a)$. Specifically, [Yor03, YRSW07], gives an algorithm for $\hat{\gamma}$ that characterizes canonical abstraction using first-order logic with transitive closure. Here, we use the $\hat{\gamma}$ operation to develop algorithms for the following operations on shape abstractions:

- Computing the most-precise abstract value that represents the (potentially infinite) set of states defined by a formula. We call this operation $\hat{\alpha}$ because it is a symbolic version of the algebraic operation α . Formally, $\hat{\alpha}(\varphi)$ computes $\alpha(\llbracket\varphi\rrbracket)$ where $\llbracket\varphi\rrbracket$ is the set of concrete states that satisfy φ .
- Computing the most-precise abstract value for the set of states that are represented by a and satisfy φ . We call this operation $assume[\varphi](a)$. Intuitively, $assume[\varphi](a)$ refines the abstract value a according to φ . Formally, $assume[\varphi](a)$ computes $\alpha(\llbracket\varphi\rrbracket \cap \gamma(a))$.
- Computing *best abstract transformers* for atomic program statements and conditions [CC79]. The current transformers in TVLA are conservative, but are not necessarily the best. Moreover, transformers automatically constructed by TVLA are often not precise enough for proving the properties of interest (require user-specified Focus formulas). The algorithm we propose eliminates the need for Focus and Coerce operations, thus improving the automation of TVLA.

Technically, if the concrete semantics of a statement is expressed as the formula τ over the input and output states, and a is the input abstract value, then computing the result of the best abstract transformer amounts to computing $assume[\tau](a)$, and projecting. Similarly, we can compute best transformers for loop-free code fragments (i.e., blocks of atomic program statements and conditions).

In a similar way, we can compute the effect of a procedure call, given the procedure’s specification, and therefore perform modular shape analysis. This is perhaps the most exciting application of the method, because it would permit TVLA to be applied to large programs by using procedure specifications.

- Computing the most-precise overapproximation of the meet of two abstract values. Such an operation provides a natural way of handling conditional statements. Furthermore, this operation is useful for combining

³As a convention, a name of an abstract operation marked with a “hat” ($\hat{}$) denotes the corresponding symbolic operation.

Operation	Meaning	Algorithm
$\hat{\alpha}(\varphi)$	$\alpha(\llbracket\varphi\rrbracket)$	See Section 3.2
$assume[\varphi](a)$	$\alpha(\llbracket\varphi\rrbracket \cap \gamma(a))$	$\hat{\alpha}(\varphi \wedge \hat{\gamma}(a))$
Best transformer of τ and a	$(\alpha \circ \llbracket\tau\rrbracket \circ \gamma)(a)$	see Section 3.3
Meet of a_1 and a_2	$\alpha(\gamma(a_1) \cap \gamma(a_2))$	$\hat{\alpha}(\hat{\gamma}(a_1) \wedge \hat{\gamma}(a_2))$
Domain change from A to B	$\alpha_B(\gamma_A(a))$	$\hat{\alpha}_B(\hat{\gamma}_A(a))$

Figure 1.3: Symbolic algorithms for most-precise abstract operations for shape analysis.

forward and backward shape analysis to establish temporal properties, and when performing interprocedural analysis in the Sharir and Pnueli functional style [SP81]. An algorithm for computing an overapproximation of the meet operation for canonical abstraction is described in [AMSS06].

Technically, the meet operation of abstract values a_1 and a_2 computes $\alpha(\gamma(a_1) \cap \gamma(a_2))$.

- Computing the most-precise abstract value in the abstract domain B for the set of concrete states represented by the abstract value a from some other abstract domain A . This domain-change operation is useful in modular analysis when different parts of the program are analyzed using different abstractions.

Technically, the domain change operation computes $\alpha_B(\gamma_A(a))$, where α_B and γ_A denote the abstraction and the concretization functions for the corresponding domains.

The core algorithm described in Section 3.2 implements the $\hat{\alpha}$ operation. Fig. 1.2(b) depicts the idea behind the algorithm. We use X to denote $\llbracket\varphi\rrbracket$, i.e., the set of concrete states that satisfy φ . The algorithm works its way down in the abstract domain, which on the left corresponds to progressing from the outer oval towards the inner region, labeled X . The algorithm repeatedly refines the intermediate abstract value by eliminating the ability to represent concrete states that are not in X . At every point, the intermediate abstract state is sound, because it always represents all the states of X . Therefore, even if the algorithm is stopped prematurely, the intermediate result can be used to check safety properties (unlike the algorithm described in Section 1.3.1).

Properties of the Algorithm Under certain conditions, detailed later, the algorithm produces an abstract value that represents the tightest set of concrete states that contains X and expressible in the abstract domain. Of course, because not all sets of concrete states are expressible in the abstract domain, the result may also represent states outside of X . The $\hat{\alpha}$ algorithm requires a theorem prover for the logic that is expressive enough to characterize the canonical abstraction, as discussed in Section 1.3.4. Even if a theorem prover call is not conclusive, our algorithm will still produce an abstract value that represent a superset of X , but we will lose the ability to guarantee that it represents the tightest superset of X .

Using $\hat{\alpha}$ and $\hat{\gamma}$ operations, we can implement *assume* and other operations, as shown in Fig. 1.3. The algorithms are obtained from the meaning of the operations by replacing the abstract operators with their symbolic counterparts. It is straightforward to see the correctness of these algorithms, assuming the correctness of $\hat{\gamma}$ and $\hat{\alpha}$. The implementation of the best abstract transformer is slightly more involved due to the use of both input and output states, and it is explained in greater detail in Section 3.3.

In practice, using a direct algorithm for $assume[\varphi](a)$ is more efficient than implementing it as $\hat{\alpha}(\varphi \wedge \hat{\gamma}(a))$, because the direct algorithm can start the iterative process of refinement from the value a instead of \top , thus avoiding some of the refinement steps. Moreover, if we have a direct algorithm for *assume*, we can also implement $\hat{\alpha}(\varphi)$ as $assume[\varphi](\top)$. Therefore, having an algorithm for either *assume* or $\hat{\alpha}$, we can implement all other operations mentioned above. In Section 3.2 we give direct algorithms for both *assume* and $\hat{\alpha}$.

This material is described in detail in Chapter 3. This chapter is largely based on the material originally published in [YRS04]. In addition to the material published in [YRS04], Appendix B of this thesis contains a formal proof of correctness of the algorithm.

1.3.3 Comparison between the Algorithms

We summarize the similarities and the differences between the two algorithms described above. The first one is described in Section 1.3.1 (details in Chapter 2), and the second in Section 1.3.2 (details in Chapter 3).

- Both algorithms use a process of iterative refinement, employing a theorem prover in each step, but they operate differently. The first algorithm works its way up the abstract domain, increasing the coverage of X in each iteration. The second algorithm works its way down the abstract domain, eliminating concrete states that are not in X .
- For both algorithms, if all theorem prover calls are conclusive and the algorithm terminates, its result is the same as the result of the most-precise abstract interpreter for the given abstraction.
- If the first algorithm is stopped prematurely, then the results might be unsound (i.e., might miss errors) or imprecise (i.e., might produce false alarms). For the second algorithm, the intermediate results are always sound, and thus they can be used for verification even if the algorithm terminated prematurely, but we lose the precision guarantees.
- For both algorithms, if a theorem prover or a model generator fails, as discussed in Section 1.3.1, we can use standard techniques to guarantee that the result will be sound if the algorithm eventually terminates. However, for the first algorithm, we lose the ability to guarantee precision or termination (sometime both, depending on the strategy that we use to recover from the failure of the model generator, see Section 2.4.5). For the second algorithm, we only lose the precision guarantee, but not termination.
- Both algorithms rely on a validity checker for certain formulas. In addition, the first algorithm requires a model generator, while the second algorithm does not.

In practice, it is not clear whether this difference makes the second algorithm easier to use than the first. On one hand, it might be difficult to find a model generator, either as part of the theorem prover or as a separate tool. Moreover, model generation might require a significant amount of time and resources. On the other hand, model generation enables the use of concrete execution (from fabricated states), which can reduce the number of iterations of the algorithm and thus, the number theorem prover and model generation calls.

- The first algorithm is applicable to any (possibly infinite) abstract domain with a finite height. The second algorithm is specialized for canonical abstraction [SRW02].⁴ However, the canonical abstraction, being parametric, is applicable to many interesting problems, in particular, reasoning about linked data-structures, which is the main application of this thesis.
- Both algorithms can be used for modular analysis, because they provide a way to compute the effect of a procedure call from a procedure's specification.

1.3.4 The Role of a Theorem Prover

The success of the algorithms described above depends on having an automatic tool that can check validity of certain formulas and generate concrete counterexamples for invalid formulas. Technically, there are off-the-shelf automatic theorem provers that can be used, e.g., SPASS [Wei], Vampire [RV01], Simplify [DNS03], Zap [BLM05], Darwin [BFT05]. Unfortunately, most such theorem provers do not produce concrete counterexamples for invalid formulas (with the exception of Darwin [BFT05]). Instead, a separate tool for model generation can be used (e.g., Paradox [CS03]).

The following difficulties arise when using theorem provers and model generators:

- The theorem prover might fail to prove validity of a (valid) formula (e.g., Simplify [DNS03] might return “invalid” for a valid formula with quantifiers).
- The theorem prover might timeout without a conclusive answer, because it exceeds the time or the amount of resources allocated for it.
- The model generator might fail to produce a counterexample (e.g., because the formula is, in fact, valid, but the theorem prover failed to prove its validity, or even when the formula is invalid).

For certain abstractions, the queries posed by our algorithms can be expressed in a decidable logic, which guarantees a (terminating) decision procedure. In practice, a decision procedure might also fail to give a conclusive answer within a reasonable amount of time. Even if the theorem prover or the model generator fail for one of the reasons above, our algorithm can use standard techniques to guarantee that the result is sound, while losing the ability to guarantee precision or termination.

⁴Extending the second algorithm beyond canonical abstraction is a subject of an ongoing work.

The algorithms spend a significant amount of time in checking validity and model generation. Moreover, failure of these tasks for one of the reasons above might cause loss of precision or termination guarantees. The choice of a theorem prover and the model generator depends on the queries posed by the algorithms, which can include the following components:

- user-provided procedure specifications and assertions,
- characterization of abstract domain (via $\hat{\gamma}$ operation),
- strongest postconditions

Therefore, it is crucial to choose a theorem prover and a model generator that match the expressive power of the abstraction, the properties of interest, and the semantics of basic program statements. For our main application, namely shape analysis, it boils down to expressing properties of linked data-structures, that often involve reasoning about reachability between elements of a data-structure. The problem is that automatic theorem provers usually do not support reachability.

A natural formalism to specify properties involving reachability is the first-order logic over graph structures with transitive closure. Unfortunately, even simple decidable fragments of first-order logic become undecidable when transitive closure is added [GME99, IRR⁺04a]. While first-order logic is also undecidable, there are many automatic theorem provers that can be useful for certain problems.

One approach to handling reachability is to harness existing theorem provers for first-order logic. By providing sufficient axiomatization [LAIR⁺05], we can, in some cases, automatically prove properties that involve the (absence) of reachability. However, in general, there cannot be a complete, recursively-enumerable axiomatization of transitive closure [Avr03, LAIR⁺05].

We take an alternative approach, and develop a formalism that (i) can express relevant properties (invariants) of various kinds of linked data-structures, including temporal violations of data-structure invariants, and (ii) has the closure and decidability features needed for automated verification. The aim of the work described in the next section is to study such a formalism based on logics over arbitrary graph structures, and to find a balance between expressiveness, decidability and complexity.

1.3.5 The Logic of Reachable Patterns

Reachability is a crucial notion for reasoning about linked data-structures. For instance, to establish that a memory configuration contains no garbage elements, we must show that every element is reachable from at least one program variable. Other examples of properties that can be naturally modeled using reachability are (1) data-structure invariants, e.g., the tail of a queue is reachable from the head of the queue, (2) the acyclicity of data-structure fragments, i.e., every element reachable from node u cannot reach u , (3) the property that a data-structure traversal terminates, e.g., there is a path from a node to a sink-node of the data-structure, (4) the property that, for programs with procedure calls when references are passed as arguments, elements that are *not* reachable from an actual parameter are not modified.

In this work, we propose a logic that can be seen as a fragment of the first-order logic with transitive closure. Our logic (1) is simple and natural to use, (2) is expressive enough to cover important properties of a wide class of arbitrary linked data-structures, and (3) allows for algorithmic modular verification using programmer-specified loop-invariants, preconditions, and postconditions.

Alternatively, our logic can be seen as a propositional logic with atomic propositions (called reachability constraints) modelling reachability between heap objects pointed-to by program variables and other heap objects with certain properties. The properties are specified using *patterns* that limit the neighborhood of an object.

For example, we can specify the property that an object v is an element of a doubly-linked list using the pattern $inv_{f,b}$, defined by $(v \xrightarrow{f} w) \Rightarrow (w \xrightarrow{b} v)$. This pattern says that if v has an emanating forward pointer f that leads to an object w , then w has a backward pointer b into v . Using the pattern $inv_{f,b}$, we can describe a doubly-linked list pointed-to by a program variable x by the atomic proposition $x[\xrightarrow{f}^*]inv_{f,b}$ in our logic. This reachability constraint says that any object v reachable from an object pointed-to by x using a (possibly empty) sequence of forward pointers satisfies the property $inv_{f,b}$.⁵

The design of our logic is guided by the following principles. First, reachability constraints are closed formulas without quantifier alternations. This guarantees that we are dealing with alternation-free formulas. Second, reachability is expressed via the Kleene star operator. We believe that regular expressions yield a more user-

⁵This and other examples are explained in detail in Section 4.3.2.

friendly notation than the transitive closure operator. Third, decidability is obtained by syntactically restricting the way patterns are formed. In particular, the use of equality is limited. Semantically, the restriction means that a pattern cannot relate two nodes that are distant from one another, unless these nodes are “named”. As a result, a pattern can only describe local properties. Global properties can only be described using reachability along regular paths that start from “named” nodes. Therefore, complex properties can be enforced only between “named” nodes. For example, complex sharing patterns can be created around objects pointed-to by program variables; arbitrary sharing is allowed but cannot be enforced deep in the data-structure, because the objects that are deep are indistinguishable and distant nodes cannot be related by a pattern.

The contributions of this work can be summarized as follows:

- We define the logic \mathcal{L}_0 where reachability constraints such as those mentioned above can be used. Patterns in such constraints are defined by (restricted) quantifier-free first-order formulas over graph structures and sets of access paths are defined by regular expressions.
- We show that \mathcal{L}_0 has a finite-model property, i.e., every satisfiable formula has a finite model. Therefore, invalid formulas are always falsified by a finite graph structure.
- We prove that the logic \mathcal{L}_0 is undecidable.
- We define restrictions on the patterns which lead to a fragment of \mathcal{L}_0 called \mathcal{L}_1 .
- We prove that the satisfiability and validity problems of \mathcal{L}_1 -formulas are decidable. The fragment \mathcal{L}_1 is the main technical result of this work and the decidability proof is non-trivial. The main idea is to show that every satisfiable \mathcal{L}_1 formula is also satisfied by a tree-like graph. Thus, even though \mathcal{L}_1 expresses properties of arbitrary data-structures, the syntax of the logic is limited enough to ensure that a formula that is satisfied on an arbitrary graph is also satisfied on a tree-like graph. Therefore, it is possible to answer satisfiability (and validity) queries for \mathcal{L}_1 using a decision procedure for weak monadic second-order (MSO) logic on trees.
- We show that despite the restriction on patterns we introduce, the logic \mathcal{L}_1 is still expressive enough for use in program verification: various important data-structures, and loop invariants concerning their manipulation, are in fact definable in \mathcal{L}_1 .
- We show that the proof of decidability of \mathcal{L}_1 holds “as is” for many useful extensions of \mathcal{L}_1 .

We define *The Logic of Reachable Patterns* (*LRP* for short) to be one of the decidable extensions of \mathcal{L}_1 (see Section 4.8 for details). For instance, in contrast to decidable logics that restrict the graphs of interest (such as weak monadic second-order logic on trees), our logic allows arbitrary graphs with an arbitrary number of fields. We show that this is very useful even for verifying programs that manipulate singly-linked lists in order to express postconditions and loop invariants that relate the input and the output state.

Our logic is expressive enough to encode many interesting data-structure invariants and loop invariants. Note that loop invariants often describe more complex structures than those satisfying data-structure invariants. The reason is that loop invariants also capture intermediate states of high-level operations, in which the data-structure invariant may be violated. The ability to express loop invariant is important to reestablish the data-structure invariant after a sequence of low-level mutations that temporarily violate the data-structure invariant.

By restricting the syntax, we guarantee that queries posed over arbitrary graphs can be answered by considering only tree-like graphs. This approach allows us to automate the reasoning about limited but interesting properties of arbitrary graphs. Moreover, our logic strictly generalizes the decidable logic in [BRS99], which inspired our work. Therefore, it can be shown that certain heap abstractions including [Hen90, SRW98, MYRS05, LAIS06] can be expressed using *LRP* formulas.

This material is described in detail in Chapter 4. The main technical result is the proof of decidability of \mathcal{L}_1 . Part of this material was originally published in [YRS+06], and an extended version of [YRS+06] was invited for a journal publication and appeared in [YRS+07]. In addition to the material already published in [YRS+07], Section 4.7.2 of this thesis contains a proof of the upper bound on the complexity of the validity problem for *LRP*.

The methods presented in Chapters 2 and 3 use abstraction and automatically infer loop invariants. In Chapter 4, we motivate the design of a new logic, *LRP*, using examples of classical verification with user-provided loop invariants. In addition, we show (in Section 4.3.4) that *LRP* is expressive enough to characterize, via $\hat{\gamma}$, certain shape abstractions. Therefore, *LRP* can also be used with the algorithms described in Chapters 2 and 3 for inferring loop invariants that involve properties of linked data-structures.

Chapter 2

Combining Concrete Execution, Abstraction, and Theorem Proving

In this chapter, we present a method for static program analysis that leverages tests and concrete program executions. State abstractions generalize the set of program states obtained from concrete executions. A theorem prover then checks that the generalized set of concrete states covers all potential executions and satisfies additional safety properties. Our method finds the same potential errors as the most-precise abstract interpreter¹ for a given abstraction and is potentially more efficient. Our method makes the process of designing new program analyses easier and more automatic, because we do not require abstract transformers. Additionally, it provides a new way to tune the performance of the analysis by alternating between concrete execution and theorem proving.

To evaluate the feasibility of our technique, we have implemented two prototypes: the first prototype is based on the XRT model checker infrastructure as its platform and uses predicate abstraction; the second prototype is based on the TVLA system as its platform and uses canonical abstraction.

The material described in this chapter, was originally published in [YBS06] and an extended version [YBS06] was invited for a journal publication in [YBS07]. This material is closely related to, and was inspired by, an earlier work [RSY04], as explained in Section 2.7.

2.1 Introduction

Recently, there has been much interest in combining dynamic and static methods for analyzing programs [NE02, GKS05, CS05, PPV05, GHK⁺06]. Dynamic analysis (or testing) is based on concrete program executions and *underapproximates* the set of program behaviors. That is, if B_P denotes the set of all behaviors of a program P then dynamic analysis explores a finite subset of B_P . Static analysis is based on an abstract interpretation [CC77] of program behavior and typically *overapproximates* the set of program behaviors. That is, static analysis has the effect of analyzing a superset of B_P , which may include *infeasible* behaviors that cannot be exhibited by the program.

The pros and cons of the two techniques are clear. If dynamic analysis detects an error then the error is real. However, dynamic analysis cannot provide a proof of the absence of errors. On the other hand, if static analysis does not find an error (of a particular kind) in the superset of B_P then B_P clearly cannot contain an error (of that same kind). However, if static analysis detects an error, it may be a false error as the behavior that induces the error may lie outside B_P .

We show how to perform static analysis using a novel combination of dynamic analysis, abstraction, and an automated theorem prover. Our technique is oriented towards finding a proof rather than detecting real errors. As a result, it has the pros and cons of a static analysis, but leverages dynamic analysis as its execution vehicle.

Our method uses state abstractions to generalize the set of program states gathered by monitoring concrete executions of a program P . An automated theorem prover is used to check that the generalized set of concrete

¹We use the term “most-precise abstract interpreter” to refer to the abstract interpreter that uses the best abstract transformers for all (intraprocedural) statements and does not use widening.

```

foo(int x, int y) {
    int *px = NULL;
    A: x = x + 1;
    B: if (x<4)
    C:   px = &x;
    D: if (px==&y)
    E:   x = x+1;
    F: if (x<5)
    G:   *px = *px+1;
    H: return;
}

```

Figure 2.1: The procedure `foo` contains a null pointer dereference error at line G.

states covers all potential executions of P (essentially the set B_P) and satisfies additional safety properties. If this check succeeds, we have a proof that all executions of the program satisfy the given properties.

However, if this check fails, our technique creates a *fabricated* concrete state from which we continue concrete program execution. We use a model generator (a theorem prover that can produce concrete counterexamples) to create fabricated states that increase the coverage. Under some standard assumptions (detailed later) our method is guaranteed to converge and obtain the same result as a standard abstract interpretation of the program P . In particular, our method produces the same amount of false alarms as a standard abstract interpretation (over the same abstract domain). It is noteworthy that we can make this guarantee even if we prematurely halt concrete execution in order to perform the coverage check. In this way, we can control the amount of time spent executing the program vs. the amount of time spent calling the theorem prover.

Additionally, we show in Section 2.6 that our method can find safety proofs with much simpler abstractions than those used by other methods [LY92, PPV05] which combine concrete execution, abstraction and theorem proving.

Finally, this method explains the result of abstract interpretation in terms of concrete executions and abstraction. This sheds some light on the trade-offs that arise when combining dynamic and static analyses.

We implemented our method in two platforms: the XRT system [GTS05] for generating unit tests and the TVLA system [LAS00] for performing shape analyses. The former prototype, based on predicate abstraction, employs the Simplify theorem prover [DNS03] and a naïve model generator. The XRT implementation supports all C# features including pointers and procedures. The latter prototype, based on canonical abstraction, employs the Paradox model finder [CS03] and a naïve theorem prover (based on “coerce” [SRW02]). The prototypes demonstrate the feasibility of our approach for two different abstractions, but not yet engineered to perform modular program analysis.

The remainder of this chapter is organized as follows. Section 2.2 illustrates the method using a simple example. Section 2.3 formalizes our method using the terminology of abstract interpretation. Section 2.4 discusses some of the practical issues that arise when implementing the algorithm. Section 2.5 describes the two prototype implementations: one based on the XRT infrastructure, and one based on the TVLA system. Section 2.6 demonstrates, using the example of Bakery mutual exclusion protocol, that our method avoids unnecessary refinements, compared to other methods that combine dynamic and static analysis. Section 2.7 further compares our approach to related work. Finally, Appendix A reviews standard definitions, and contains the proofs of all theorems from Section 2.3.

2.2 Example

We now illustrate our basic method using a simple example, shown in Fig. 2.1. The procedure `foo`, written in C syntax, contains a null pointer dereference error at line G. A concrete state of this procedure is described by a quadruple (pc, i, j, ptr) , where pc is the value of the program counter ranging over the labels A–H, i and j are the integer values of variables x and y , and ptr is the value of `px` of type integer pointer. In the rest of this example,

we abuse the notations slightly and use x to refer to the program variable and to its value. Similarly for y .

To simplify the exposition, we assume that `foo` can be called with any integer values for the variables x and y , which defines the set of all possible initial states.

When `foo` is called with $x = 3$, the value of px is `NULL` on the left-hand side of the assignment statement at label `G`, causing a null pointer dereference at `G`. Note that the conditional at label `D` always evaluates to false, so the assignment statement at label `E` is dead code.

We use the following abstraction function: for every set of concrete states X ,

$$\alpha(X) = \{(pc, x < 5, px = NULL) \mid (pc, x, y, px) \in X\}$$

Here, a concrete program state is mapped to a triple of values for the following expressions: the program counter pc (ranging over eight labels), the predicate $(x < 5)$ and the predicate $(px = NULL)$. A predicate evaluates to true (t) or false (f). For a singleton set C , we abuse the notations slightly by writing $\alpha(pc, x, y, px)$ instead of $\alpha(\{(pc, x, y, px)\})$.

The abstract state space is finite, and it consists of $8 \times 2 \times 2 = 32$ possible triples. The reachable abstract states are shown in Fig. 2.2(a). Our method does not construct these abstract states beforehand. Instead, we execute the procedure on a set of tests and compute, using α , the abstraction of the concrete states encountered during test executions.

Now consider executing `foo(2, 0)`, `foo(6, 0)`, `foo(11, 0)`, that define the test set $T = \{(A, 2, 0, NULL), (A, 6, 0, NULL), (A, 11, 0, NULL)\}$. The test set T does not uncover the null-pointer dereference in the program. The abstract states covered by the execution of T , denoted by A_T , are marked in Fig. 2.2(a) with bold-circles. Note that the error abstract state (G, t, t) is not in A_T .

2.2.1 Example of Finding a Bug

Next, we check whether the test set T is adequate under α . That is, is the set of covered abstract states A_T an invariant? This check fails, because there is a concrete state b such that $\alpha(\{b\}) = (B, t, t)$, a covered abstract state, from which in one step of the program it is possible to reach a concrete state d such that $\alpha(\{d\}) = (D, t, t)$, an uncovered abstract state. Using a model generator, our method fabricates such a pair of concrete states, say $b = (B, 4, 0, NULL)$ and $d = (D, 4, 0, NULL)$. Execution from state b leads to a null pointer dereference error at label `G`, as shown in Fig. 2.2(a). Thus, our analysis reports a potential error. In this case the program contains a real error, because the state b is a reachable concrete state (reachable from the initial state $(A, 3, 0, NULL)$).

2.2.2 Example of Finding a Proof

Now, let us consider what our technique does on a version of the above procedure obtained by modifying the conditional at label `B` from $(x < 4)$ to $(x < 5)$, which eliminates the null pointer dereference. Let us call the new version `fixed_foo`. The abstract state space of `fixed_foo`, obtained using the abstraction function α as before, is shown in Fig. 2.2(b).

Again, the set of covered abstract states is not an invariant. In particular, there is a concrete state d' such that $\alpha(\{d'\}) = (D, t, f)$ from which in one step of the program it is possible to reach a concrete state e' such that $\alpha(\{e'\}) = (E, t, f)$, an uncovered abstract state. A pair that satisfies this constraint is $d' = (D, 4, 0, \&y)$ and $e' = (E, 4, 0, \&y)$. Note that neither of these states is a reachable concrete state, as the address of the variable y is never assigned to the variable px in the program, and thus the label `E` is not reachable. However, in the abstract state space, the label `E` is reachable whenever the predicate $(px = NULL)$ is false at label `D`.

Concrete execution from the state d' covers the additional abstract states: (E, t, f) , (F, f, f) , and (H, f, f) . At this point, our method shows that the set of covered abstract states is an invariant. This implies that the abstract states represent all reachable concrete states of `fixed_foo`. Since these abstract states do not contain the error state (G, t, t) , we have found a proof that there cannot be any null pointer dereference at label `G`, in any execution of the procedure `fixed_foo`.

Interestingly, (H, f, f) represents some reachable concrete states, i.e., there exists a test input (not in our test set) that covers (H, f, f) . However, finding such an input is a non-trivial task, because to cover (H, f, f) `fixed_foo` must be called with precisely $x = 3$ as an argument (and any value of y). Random or symbolic path-exploration techniques can be used to address the problem, but we avoid this problem using fabrication. The

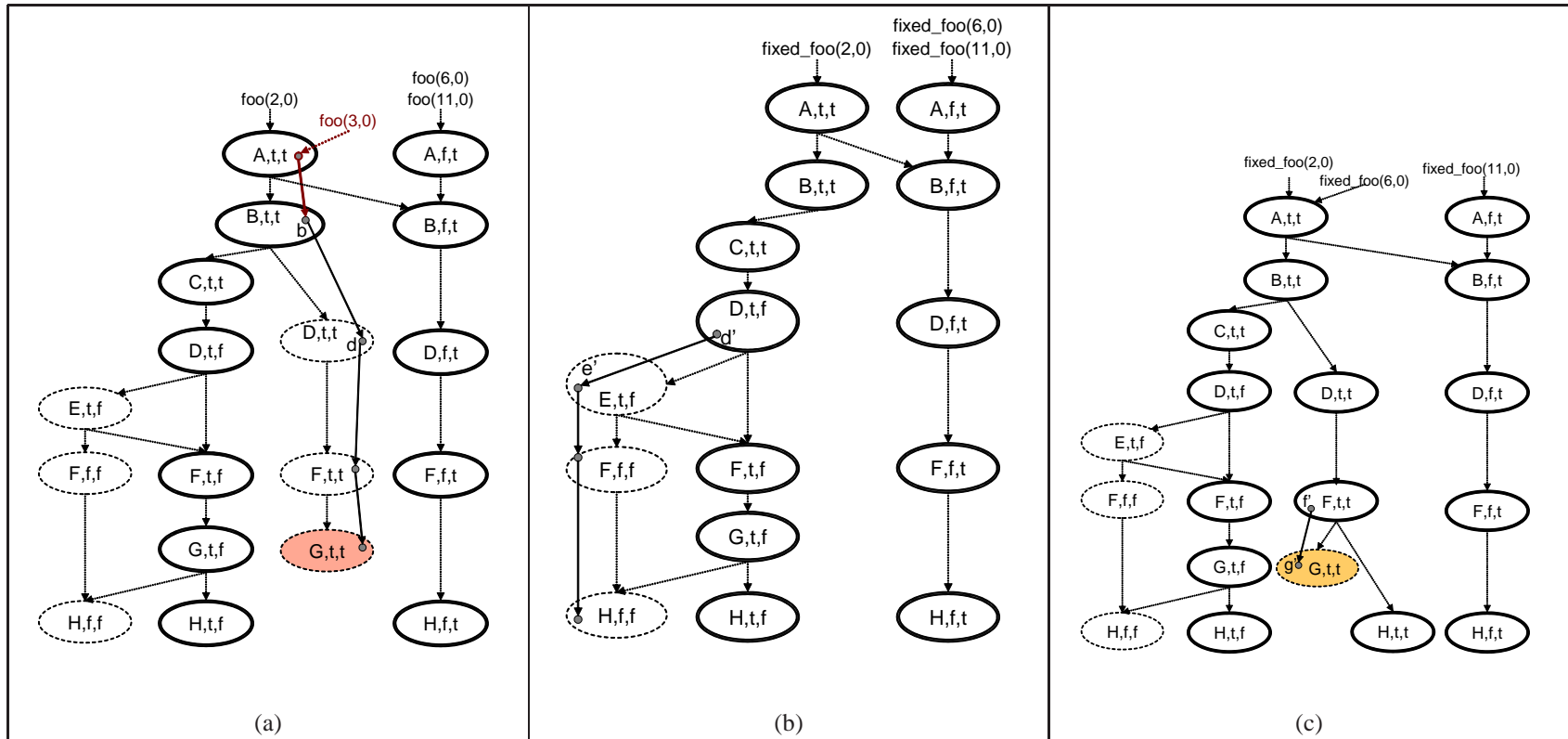


Figure 2.2: Reachable abstract states for (a) finding a null pointer dereference in `foo` using $\alpha(pc, x, y, px) = (pc, x < 5, px = NULL)$; (b) finding a proof for `fixed_foo` using $\alpha(pc, x, y, px) = (pc, x < 5, px = NULL)$; (c) finding a false error in `fixed_foo` using $\alpha'(pc, x, y, px) = (pc, x < 10, px = NULL)$. Abstract states covered by the set of tests $T = \{(A, 2, 0, NULL), (A, 6, 0, NULL), (A, 11, 0, NULL)\}$ are marked with bold-circles.

abstract state (H, f, f) is covered by a test execution that starts from the fabricated state d' . It shows that we can benefit from fabricated states to discover abstract states that are reachable via rare executions.

Moreover, execution of d' covers the abstract state (F, f, f) , which does not represent any reachable concrete state (although the statement at label F is reachable). Fortunately, all executions from this abstract state are safe. That is, the current abstraction shows the absence of errors in all feasible executions as well as in some infeasible executions. This shows the strength of our method: we obtain a proof using a much coarser abstraction than the existing methods [PPV05, Bal04] that are based only on feasible executions. In this example, these methods would unnecessarily refine the abstraction, even though the current abstraction is sufficient to prove the absence of null pointer dereferences (and our method obtains a proof).

2.2.3 Example of Finding a False Error

Now, let us show an abstraction function that is not precise enough to prove the correctness of the (corrected) program `fixed.foo` above. Suppose our abstraction function is:

$$\alpha'(C) = \{(pc, x < 10, px = NULL) \mid (pc, x, y, px) \in C\}$$

In this case, the abstraction function cannot precisely track the relationship between the value of x and the predicate $(px = NULL)$ in the program. Fig. 2.2(c) shows the reachable abstract states for this new abstraction function.

Using a theorem prover and a model generator, we find a concrete state f' such that $\alpha'(\{f'\}) = (F, t, t)$ from which in one step of the program it is possible to reach a concrete state g such that $\alpha'(\{g\}) = (G, t, t)$, an uncovered abstract state. A pair that satisfies this constraint is $f' = (F, 4, 0, NULL)$ and $g = (G, 4, 0, NULL)$. Note that f' is not a reachable concrete state. Running the program from state f' will cause a null pointer dereference to occur at label G . At this point, our analysis reports a potential error. This false error would also be reported by an abstract interpreter using the abstraction function α' .

Our analysis does not distinguish between a false error such as this one, and a real error such as the one in Section 2.2.1. Both are reported as *potential* errors. In particular, our method might not discover the test input $(A, 3, 0, NULL)$ mentioned in Section 2.2.1.

2.3 Formal Description

This section formalizes our method and compares it to the traditional static analysis by abstract interpretation. Section 2.3.1 quickly reviews relevant terminology about abstract interpretation. Section 2.3.2 presents an idealized version of our method and discusses its basic properties. It does not provide an effective algorithm, as it uses an incomputable operation. Section 2.3.3 discusses how to realize the method as a symbolic algorithm which employs a theorem prover and a model generator.

2.3.1 Abstraction and Concretization

Let \mathcal{C} be a set of concrete states of program P (not necessarily reachable). Let \mathcal{A} be a set of abstract values (not necessarily reachable).

In abstract interpretation, we usually assume that \mathcal{A} forms a lattice, with partial order \sqsubseteq , meet \sqcap and join \sqcup operations (see Appendix A.1). In the previous section, we used a powerset lattice, in which an element (abstract value) is a set of abstract states, ordered by set inclusion \subseteq , meet is set-intersection \cap , and join is set-union \cup .

An **abstraction function** $\alpha: 2^{\mathcal{C}} \rightarrow \mathcal{A}$ yields an abstract value that represents a set of concrete states. A **concretization function** $\gamma: \mathcal{A} \rightarrow 2^{\mathcal{C}}$ yields a set of concrete states that an abstract value represents. The partial order on \mathcal{A} satisfies for all $a, a' \in \mathcal{A}$,

$$a \sqsubseteq a' \Rightarrow \gamma(a) \subseteq \gamma(a'). \quad (2.1)$$

The concretization and the abstraction functions form a **Galois connection** between $2^{\mathcal{C}}$ and \mathcal{A} , i.e., for all $a \in \mathcal{A}$ and $X \subseteq \mathcal{C}$:

$$\alpha(X) \sqsubseteq a \iff X \subseteq \gamma(a) \quad (2.2)$$

```

[1] procedure basic( $T_0$ )
[2]    $a := \perp$ 
[3]    $T := T_0$ 
[4]   while(true) begin
[5]      $C := \text{Execute}(f, T)$ 
[6]      $a := a \sqcup \alpha(C)$ 
[7]     if exists  $\sigma \in f(\gamma(a))$  s.t.  $\sigma \notin \gamma(a)$ 
[8]       then  $T := \{\sigma\}$ 
[9]     else return  $a$ 
[10] end

```

Figure 2.3: The basic procedure. Here, $T_0, T, C \subseteq \mathcal{C}$, and $a \in \mathcal{A}$. If $\alpha(T_0) = \alpha(I)$, then the result of the procedure is a sound approximation of P .

This implies that $X \subseteq \gamma(\alpha(X))$ and $\alpha(\gamma(a)) \sqsubseteq a$. That is, abstraction followed by concretization potentially yields more states, and concretization followed by abstraction potentially yields a more precise abstract value. Also, it follows from (2.2) that α uniquely determines γ .

Given an abstraction function α , it is easy to define the corresponding concretization function: $\gamma(a) = \{c \mid \alpha(\{c\}) \sqsubseteq a\}$. For example, using the abstraction function α' from Section 2.2.3, $\gamma(\{(G, f, t)\}) = \{(G, x, y, NULL) \mid x \geq 10\}$. Note that the set $\gamma(\{(G, f, t)\})$ is infinite and the values of y are not restricted.

To simplify the presentation, we assume that an abstract value in \mathcal{A} collectively describes states at all program points, rather than having a separate abstract value for each program counter. This can be achieved by encoding the program counter in the representation of a concrete state \mathcal{C} , as we did in the previous section.

The program P defines a transition relation on concrete states $\rightarrow_P: \mathcal{C} \times \mathcal{C}$. For $\sigma, \sigma' \in \mathcal{C}$, we say that σ' is a successor of σ if $\sigma \rightarrow_P \sigma'$. Intuitively, σ' is a result of executing a single statement of P in the state σ . We define the function $f_P: 2^{\mathcal{C}} \rightarrow 2^{\mathcal{C}}$ as follows:

$$f_P(X) = \{\sigma' \mid \sigma \rightarrow_P \sigma', \sigma \in X\} \cup \{X\}$$

Note that f_P is monotone and extensive (i.e., $X \subseteq f_P(X)$). We drop the subscript P when it is understood from the context. The set of concrete states reachable from $X \subseteq \mathcal{C}$ is the least fixed point of f w.r.t. X , denoted by $\text{LFP}_{\sqsubseteq X}(f)$.

Let $I \subseteq \mathcal{C}$ be the set of all possible initial states of a program P . The meaning of the program P is the set of all concrete states reachable from some initial state: $\text{LFP}_{\sqsubseteq I}(f)$.

An abstract value $a \in \mathcal{A}$ is a **sound** overapproximation of P if a represents all concrete states reachable from I (but possibly other states):

$$\text{LFP}_{\sqsubseteq I}(f) \subseteq \gamma(a). \quad (2.3)$$

An abstract value $a \in \mathcal{A}$ is **invariant under** P if

$$f(\gamma(a)) \subseteq \gamma(a) \quad (2.4)$$

Theorem 2.3.1 (Soundness) *If an abstract value $b \in \mathcal{A}$ is invariant under P and $I \subseteq \gamma(b)$ then b is a sound overapproximation of P .*

2.3.2 Basic Procedure

Fig. 2.3 shows a high-level description of our method. Implementation details are discussed in the next sections. We assume that the basic procedure is called with a finite set T_0 of initial states ($T_0 \subseteq I$), such that $\alpha(T_0) = \alpha(I)$.

Line [5] corresponds to the *Execute* step, described in Section 1.3.1. Formally, $\text{Execute}(f, T)$ returns a finite subset of the states reachable from T using f that contains at least the states in T :

$$\text{Execute}(f, T) \subseteq \text{LFP}_{\sqsubseteq T}(f) \text{ and } T \subseteq \text{Execute}(f, T). \quad (2.5)$$

Note that it is not necessary (and sometimes impossible) to collect all states reachable from T . In particular, this step allows us to handle non-terminating executions or very long running executions. We require that *Execute* terminates (and can always guarantee it, for example using a timeout).

In line [6] the abstraction of the obtained concrete states is computed using α . This corresponds to the *Abstract* step described in Section 1.3.1. The procedure terminates when it is not possible to fabricate a state σ that satisfies the condition in line [7], i.e., a is invariant under P . This implies that a is a sound approximation of the reachable states of P (by Theorem 2.3.1).

Furthermore, the procedure computes the same abstract value for P as is computed by the most-precise abstract interpreter for the given abstraction, as stated by the following theorem:

Theorem 2.3.2 *Let $f^{\natural} : \mathcal{A} \rightarrow \mathcal{A}$ be defined by $f^{\natural} = \alpha \circ f \circ \gamma$. The procedure in Fig. 2.3 computes the least fixed point of f^{\natural} w.r.t. $\alpha(I) : LFP_{\sqsupseteq \alpha(I)}(f^{\natural})$.*

The particular choice of a fabricated concrete state in line [7] does not affect the final result of the procedure, but it may affect the number of iterations needed to find the result, as explained below.

Let S_a be the set of all possible fabricated states for an abstract value a , i.e., the set of states σ that satisfy the condition in line [7]:

$$S_a \stackrel{\text{def}}{=} \{ \sigma \mid \sigma \in f(\gamma(a)), \sigma \notin \gamma(a) \}$$

In line [8], a single fabricated state is chosen from S_a . It is easy to modify the procedure to work with several fabricated states together in the same iteration. That is, we can choose T to be any finite non-empty subset of S_a . We cannot use the entire set S_a because it may be infinite. Using several fabricated states at once may increase coverage more than with a single state, but it might increase the cost of concrete execution. More importantly, it might be costly to fabricate states (see Section 2.4.5).

Because fabricated states are not covered by the abstract values collected so far, the coverage strictly increases in successive iterations.

Theorem 2.3.3 *If the lattice \mathcal{A} has a finite height, the procedure in Fig. 2.3 terminates.*

Remark. If the lattice \mathcal{A} admits infinite ascending chains (e.g., polyhedra [CH78]), it is possible to use standard *widening* techniques to enforce and accelerate termination of our procedure, sacrificing the precision of its result. Let $\nabla : \mathcal{A} \times \mathcal{A} \rightarrow \mathcal{A}$ denote the widening operator on \mathcal{A} (see Appendix A.1). We replace line [6] with $a := a \nabla \alpha(C)$. The result a of the procedure satisfies $LFP_{\sqsupseteq \alpha(I)}(f^{\natural}) \sqsubseteq a$, but the result a may be less precise than $LFP_{\sqsupseteq \alpha(I)}(f^{\natural})$.

Choosing Fabricated States

Choosing any state in S_a strictly increases the coverage, but some states increase the coverage more than others. Intuitively, we would like to choose a fabricated state that covers as many new abstract states as possible, “jumping” higher in the abstract lattice. A good choice of fabricated states reduces the number of the iterations of the procedure, and thus, the number of calls to a theorem prover and a model generator.

Example 2.3.4 *Consider the following code fragment:*

```

while(x < 1000) {
A:  if (x % 2 == 0)
B:    x += 2;
    else
C:    x += 1;
}
```

The abstraction function is $\alpha(C) = \{ (pc, x > 0) \mid (pc, x, \dots) \in C \}$. Suppose that the current abstract value is $a = \{ (pc = A, t) \}$. If we choose a fabricated state at program point A with an even value of x , we cannot cover abstract states with $pc = C$. If we choose a fabricated state with an odd value of x , we cover both $pc = B$ and $pc = C$ with an execution from the same fabricated state.

Formally, we define a partial order on fabricated states: for every $\sigma, \sigma' \in S_a$, the state σ covers more abstract states than σ' , denoted by $\sigma' \preceq \sigma$, when

$$a \sqcup \alpha(\text{Execute}(f, \{ \sigma \})) \sqsubseteq a \sqcup \alpha(\text{Execute}(f, \{ \sigma' \}))$$

A finite set of fabricated states $S_a^{opt} \subseteq S_a$ is *optimal* with respect to a if $a \sqcup \alpha(\text{Execute}(f, S_a^{opt})) = a \sqcup \alpha(\text{Execute}(f, S_a))$, and for all $\sigma, \sigma' \in S_a^{opt}$, $\sigma \not\leq \sigma'$ and $\sigma' \not\leq \sigma$.

If the abstract domain has a finite height, then there exists a finite set S_a^{opt} as above. It is not clear how to find an optimal set of fabricated states, or how to approximate them, because this condition is non-local. Heuristics for choosing fabricated states are out of scope of this work.

2.3.3 Symbolic Procedure

For all abstractions we are aware of, the function α is efficiently computable given a finite set of concrete states, represented explicitly. Note that applying α in line [6] does not require the use of a theorem prover.

Nonetheless, as mentioned before, the procedure in Fig. 2.3 does not provide an effective algorithm. In particular, the γ operation used in line [7] is not computable, as $\gamma(a)$ may be infinite. We now show how to implement line [7] symbolically, using a theorem prover and a model generator.

Symbolic Characterization Concrete program states can be represented as logical structures, (e.g., constant symbols model program variables). Thus, sets of concrete states can be described by logical formulas in some logic \mathcal{L} (e.g., first-order logic). The concretization function γ can be expressed symbolically, i.e., for every $a \in \mathcal{A}$, there exists a formula in \mathcal{L} , denoted by $\hat{\gamma}(a)$, that exactly represents a : for all $\sigma \in \mathcal{C}$,

$$\sigma \models \hat{\gamma}(a) \text{ if and only if } \sigma \in \gamma(a) \quad (2.6)$$

In the example from Section 2.2, with abstraction function α , $\hat{\gamma}$ can be expressed as a quantifier-free first-order formula, interpreted over integers. The constant symbols x and y model the values of the corresponding program variables. The constant symbols A – H model each of the program points, and an additional constant symbol pc models the program counter.² Similarly, the value of the expression $px = NULL$ can be encoded with a corresponding pair of constant symbols. For instance, $\hat{\gamma}(\{(G, f, t)\})$ is the formula $(pc = G) \wedge \neg(x < 5) \wedge (px = NULL)$.

Meaning of Program Statements The meaning of a program can be expressed as a formula transformer, $\mathcal{SP}: \text{Prog} \times \mathcal{L} \rightarrow \mathcal{L}$, which defines the strongest postcondition [Dij76]: for every program $P \in \text{Prog}$ and a formula $\varphi \in \mathcal{L}$, a concrete state σ' satisfies $\mathcal{SP}(P, \varphi)$ if and only if there exists a concrete state σ such that σ' is a successor state of σ in P and σ satisfies φ . Intuitively, \mathcal{SP} describes the result of executing a single basic statement of P on a state that satisfies φ . For example, $\mathcal{SP}(x = x + 1, x > 5)$ is the formula $x > 6$.

We can also use \mathcal{SP} that describes the result of executing an entire loop-free code fragment, instead of a single basic statement. Note that we do not use \mathcal{SP} for loops, because our method automatically computes loop invariants.

Symbolically Checking for Invariance Using the $\hat{\gamma}$ and \mathcal{SP} operations, we can symbolically express the fact that abstract value $a \in \mathcal{A}$ is an invariant:

$$\mathcal{SP}(P, \hat{\gamma}(a)) \Rightarrow \hat{\gamma}(a) \quad (2.7)$$

The formula (2.7) is valid if and only if a is an invariant.³

Given a program P and an abstract value a , our method can automatically generate the formula in (2.7). Moreover, it can check the validity of (2.7) automatically using a theorem prover for \mathcal{L} . If the validity check fails, then a model generator can be used to fabricate a state that satisfies $\hat{\gamma}(a)$ and has a successor that does not satisfy $\hat{\gamma}(a)$. Formally, the fabricated state satisfies the negation of (2.7):

$$\mathcal{SP}(P, \hat{\gamma}(a)) \wedge \neg \hat{\gamma}(a) \quad (2.8)$$

For example, in Section 2.2.1, if a is $\{(B, t, t)\}$, then the formula $\hat{\gamma}(a)$ is $(pc = B \wedge x < 5 \wedge px = NULL)$. The strongest postcondition of this formula and the statement `if (x < 4)` at label `B` is the formula $sp \stackrel{\text{def}}{=} (pc = C \wedge x < 4 \wedge px = NULL) \vee (pc = D \wedge x = 4 \wedge px = NULL)$. When checking validity of $sp \Rightarrow \hat{\gamma}(a)$,

² Not every interpretation of these constants is legal; to rule out illegal interpretations of pc , the following axiom can be used: $pc = A \vee pc = B \vee \dots \vee pc = H$.

³ Alternatively, a formula based on the weakest (liberal) precondition can be used: $\hat{\gamma}(a) \Rightarrow \mathcal{WP}(P, \hat{\gamma}(a))$.

we consider only standard interpretations of integers and relations over integers. Clearly, $sp \Rightarrow \hat{\gamma}(a)$ is not valid (even in the standard interpretation). This allows us to fabricate a state, say $d = (D, 4, 0, NULL)$, such that $d \models sp \wedge \neg(pc = B \wedge x < 5 \wedge px = NULL)$.

Symbolically Checking Safety Properties In our setting, the safety properties of interest also can be expressed by a formula $\varphi \in \mathcal{L}$. For example, in Section 2.2 the safety property can be expressed by the formula $\varphi \stackrel{\text{def}}{=} \neg(pc = G \wedge px = NULL)$.

If $a \in \mathcal{A}$ is a sound approximation of P , and the formula $\hat{\gamma}(a) \Rightarrow \varphi$ is valid, then all reachable concrete states of P satisfy the safety properties φ . In Section 2.2.2, our method obtains a set of abstract states a for which $\hat{\gamma}(a) \Rightarrow \varphi$ is valid. In Section 2.2.3, one of the abstract states covered by our method is (G, t, t) , for which $(pc = G \wedge x < 10 \wedge px = NULL) \Rightarrow \neg(pc = G \wedge px = NULL)$ is not valid, and our method reports a potential error.

In practice, there are automatic tools for checking validity and generating models (even if the logic \mathcal{L} is undecidable), which have certain limitations, as discussed in the next section.

2.4 Towards a Realistic Implementation

In this section, we discuss some of the practical issues that arise when implementing the symbolic algorithm described in Section 2.3.

2.4.1 Program Analysis Infrastructure

Our method requires an infrastructure that supports: (1) monitoring of concrete program states in concrete executions to compute abstract state coverage; (2) symbolic execution of loop-free code fragments to compute \mathcal{SP} ; (3) state manipulation to create fabricated states.

Explicit-state model checkers such as SPIN [Hol03], CMC [MPC⁺02], JavaPathFinder [VHB⁺03], XRT [GTS05], which perform systematic and exhaustive testing, provide a good starting point (though not all support symbolic execution). A model checker analyzes several executions of the program at once, and controls the order in which these concrete executions advance (e.g., DFS, BFS). A model checker usually manipulates a representation of concrete states, which comes in handy for fabrication of states. Our implementation uses XRT (see Section 2.5.1).

2.4.2 Cutpoints

Section 2.3 simplifies the discussion by encoding the program counter in the abstract value. This encoding allows us to keep abstract states for each program point. In practice, it is not necessary to track all program points. We can choose a designated set of program points, called *cutpoints*, where the abstraction is computed.

The runtime overhead of computing abstract state coverage decreases when there are fewer cutpoints. Furthermore, having fewer cutpoints potentially improves the precision of our method, as the abstraction of a composition of two statements is at least as precise as a composition of their abstractions.

As in deductive verification, a minimal set of cutpoints is a set which cuts every cycle in a program's control flow graph [Flo67].

The check that an abstract value is an invariant is adapted according to the set of cutpoints: the strongest postcondition formulas \mathcal{SP} in (2.7) describe the result of executing a sequence of statements from one cutpoint to the next (and not a single statement, as before). Note that states are fabricated only at cutpoints.

2.4.3 On-the-fly Abstraction

We previously presented execution and abstraction as separate steps (line [5] and line [6] of Fig. 2.3). In practice, abstraction can be computed *on the fly* during program execution. The idea is to monitor the execution: when a cutpoint is reached, we pause the execution to compute the abstraction of the current state, conjoin the resulting abstract value with the abstract value collected so far, and continue the execution. This way, concrete states encountered during program execution need not be stored (only abstract values need to be stored).

2.4.4 Interprocedural Analysis

The simplest way to handle procedure calls in program analysis is by inlining the code of the procedure at the call site. Similarly, we can check invariants of programs with procedure calls by inlining the code of the procedure and using strongest postconditions as before. Alternatively, we can use procedure’s specification to create a formula whose validity implies that the abstract value obtained so far is an invariant. This allows us to check invariants in a modular way, and therefore enables modular program analysis with user-provided specifications.

2.4.5 Employing a Theorem Prover

The success of our method depends on having a theorem prover which can check validity of formulas from (2.7), and a model generator which can generate concrete counterexamples to validity of these formulas, as discussed in Section 1.3.4. If a theorem prover or a model generator fails, we cannot guarantee that our algorithm produces the most-precise results with respect to the given abstraction. However, we can guarantee that our algorithm, if it terminates, produces a sound result even when some theorem prover calls failed when checking that the abstract value is an invariant. First, our method attempts to fabricate a state that satisfies (2.8). If model generation succeeds, the analysis continues as before (without losing precision guarantees). Failure of the model generator to fabricate a state can be handled as follows:

- Fabricate a state outside of $\gamma(a)$. This guarantees that the coverage increases in each iteration, and the analysis eventually terminates, but it might fail to produce the most-precise result (because the fabricated state may not have a predecessor in any covered state).
- Fabricate some concrete state, say using a random generator, sacrificing both termination of the analysis and its precision. However, if the analysis terminates, its result still is sound.
- Use the hybrid approach which combines concrete execution and abstract interpretation, as discussed in Section 2.4.7.

2.4.6 Controlling Concrete Execution

Recall from Section 2.3 that stopping concrete execution at any moment does not affect termination or precision of the analysis. In fact, to guarantee termination, it is sufficient to have $C := T$ in line [5]. Normally, concrete execution is a cheap way to increase coverage, but for certain programs, concrete execution might go on for a long time without covering a new abstract state. The question is when the concrete execution should be paused to check invariance.

Our idea is to monitor how many new abstract states are covered by a concrete execution. If the coverage has not increased sufficiently for a certain period of time, the concrete execution can be paused to check invariance. If the abstract value obtained so far is not invariant, our method can increase the coverage by fabricating a new concrete state and continue concrete execution from it. In this way, we can control the amount of time spent executing the program vs. the amount of time spent calling the theorem prover.

For example, consider the code in Fig. 2.4, which uses the non-deterministic choice operation ($*$). A concrete execution that iterates through the loop 500 times, always taking the true branch of the `if` statement in line [3], and then, in the $500 + 1$ iteration, takes the false branch, reaches the assertion in line [7]. If the false branch is taken earlier, n is reset in line [8]. The assertion in line [7] fails in rare executions with a long trace to the error. Such errors are difficult to discover using an explicit-state model checker or random testing. Our approach can skip the execution of many loop iterations that do not increase coverage.

We use predicate abstraction with the predicates $n = 0$, $0 < n < 500$, $n = 500$, and $n > 500$ for value of n on line [3]. Note that the predicates divide the concrete state space into 4 partitions, as shown in Fig. 2.4(b). Model checking quickly finds concrete execution that covers the states $n = 0$, $0 < n < 500$, but then the concrete execution stays within these abstract states. At this point, our algorithm simply fabricates a state with $n = 500$, and continues model checking from it, skipping the long execution trace that leads to it. The model checker can easily find an execution through the loop body to line [7], and reports a potential error.

This example shows that switching from concrete execution to fabrication can help us in finding errors and invariants faster.

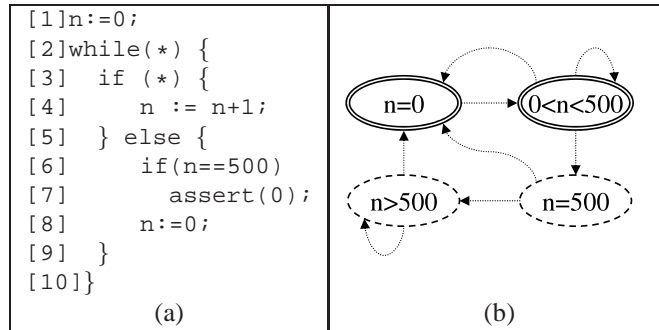


Figure 2.4: (a) Example program, (b) Abstract state-space.

2.4.7 Hybrid Approach

For certain programs, concrete execution might go on for a long time without covering a new abstract state, whereas abstract interpretation makes progress in every step, but each step can be expensive or can lose precision if the abstract transformers are not the best. An exciting application of our method is its ability to address limitations of one approach using the other, by interleaving concrete and abstract interpretation. Furthermore, abstract interpretation can be used when a model generator fails to fabricate a new state that satisfies (2.8).

Technically, the hybrid approach can choose to obtain a new abstract value by concrete execution of existing test inputs or fabricated states, as described in Section 2.3, or by applying an abstract transformer to the current abstract value, as usual in abstract interpretation. To switch from abstract interpretation mode to concrete execution, the hybrid approach symbolically checks if the new abstract value is an invariant, and attempts to fabricate states from the new abstract value, as explained in Section 2.3.3. It is possible that model generation for the current abstract value fails, but succeeds for the new abstract value, produced using an abstract transformer. The precision of the hybrid analysis depends on the precision of abstract transformers that are used, assuming that all theorem prover and model generator calls are conclusive.

2.5 Prototype Implementations

To evaluate the feasibility of the technique, we have implemented it in two prototypes: the first prototype is based on predicate abstraction and uses XRT model checker as its platform; the second prototype is based on canonical abstraction and uses TVLA system as its platform.

2.5.1 Prototype Implementation Based on XRT

We have implemented our method on top of the XRT framework [GTS05], an extensible framework for explicit and symbolic model checking of programs, represented in Microsoft’s common intermediate language (CIL). XRT processes .NET managed assemblies, and provides means for analyzing, rewriting, and executing programs. Our implementation takes advantage of all these features.

Our implementation uses predicate abstraction [GS97] (without refinement), and supports user-defined predicates. It can also automatically generate a default set of predicates by a backwards data-flow analysis from the conditional branches that infers the predicates governing these branches.

In our implementation, predicates are defined as C# methods, called *probes*. Probes return Boolean values, have no side-effects and contain no loops or method calls. Each cutpoint is instrumented to call the probe methods, which evaluate the appropriate predicates on the current state. We compute the abstraction of the concrete execution on-the-fly: the XRT runtime pauses execution immediately after a call to a probe method returns, and its return value is used to update the abstract state.

When predicates are given as logical formulas, it is easy to implement $\hat{\gamma}$ for predicate abstraction, but in our case predicates are given as CIL code. The symbolic execution of probe methods by XRT gives a natural way to construct logical formulas for predicates. The symbolic execution mode of XRT also provides strongest postconditions that our method uses to check whether the abstract value is invariant. To check validity, we use

the Simplify theorem prover [DNS03]. To fabricate states, we implemented a naïve model generator within XRT, also based on Simplify.

Our implementation places cutpoints: (i) before each loop body (to cut cycles); (ii) on entry and exit of every method, (iii) before and after every method call, and (iv) at each program point that may potentially violate safety properties, such as a pointer dereference or an array access. To handle method calls, we have implemented 0-CFL Reachability [NNH99], using the cutpoints of (ii,iii). For simplicity, we do not handle goto statements.

Unit-Testing with Fabrication

A major application of XRT is in the area of unit testing. Therefore, to evaluate the implementation of our method, we adapted the method to operate on a separate class using unit-tests for that class (rather than analyzing a closed application using its test inputs). A similar approach can be applied to analyze an open system or a component.

The idea is to invoke each method of the class on all the concrete states obtained on exit of any method of this class. We illustrate the idea on the implementation of a bounded stack, used previously in the literature [SLA02, XN03, CS04, PE05].

Example 2.5.1 *Fig. 2.5 shows an abbreviated version of the code that implements a bounded stack, using a fixed-size array.*

A bounded stack can normally be ‘empty’, ‘partially full’ or ‘full’. We used predicate abstraction to capture these states, and distinguish them from illegal states in which `size` is out of the bounds of `elems`.

The bounded stack supports the usual operations, but it does not provide any exceptional behavior. Instead, if an operation is applied in an inappropriate state, it has no effect. For example, if the stack is full, the `push` operation has no effect. However, the `pop` method incorrectly handles popping an empty stack. This problem was not exhibited by the provided unit tests, because `pop` is never called with an empty stack.

We analyzed the class using our implementation based on XRT, checking for the `IndexOutOfRangeException`. In the first iteration, our method fabricates a state σ on exit of `pop`, with an empty stack. Then, it executes the method `pop` again on the fabricated state σ , obtaining a new state σ' on the exit of `pop`, with `size` < 0 (no runtime exception occurs). Then, it executes the method `push` on σ' , causing at label L1 an `IndexOutOfRangeException`.

After fixing the error in `pop`, our analysis automatically proves absence of the `IndexOutOfRangeException` in this example, using four fabricated states, and the default predicates, as mentioned in Section 2.5.1. If the maximal size provided to the constructor is negative, it throws `OverflowException` exception, but this error should not be reported by the analysis, which tracks different exceptions.

This example shows that our analysis can deal with unexpected failures, e.g., when a concrete execution throws an exception that is not tracked by the analysis. If such exception is thrown in a concrete execution, our analysis can fabricate any state in the following program point, and continue the execution from it. This fabrication is easy because it does not place any constraints on the fabricated state. It provides a sound and (perhaps, surprisingly) most-precise result, because behavior that is not modeled is treated by a sound abstraction as if “anything can happen” at that program point.

Upon termination of the analysis, the abstract states on the entry and exit of all methods are the same. This set of abstract states, in fact, represents the class invariants, under certain conditions about the class, stated in [Log04]. The analysis can output the inferred class invariants in the form of logical formulas, by computing $\hat{\gamma}$ of the relevant abstract states.

Note that, as we are using state-based abstractions, our approach does not learn method-call order. Also, our method analyzes each class independently of the actual clients of this class. The approach may also report on potential errors, that do not occur in any actual client of this class. The advantage of this approach is that it identifies potential errors early in the development process, even before the client is written. If our method succeeds, it provides a proof of safety for the class in any client. This proof can be used to perform assume-guarantee reasoning.

Also, in the setting of unit-testing, it is easier to classify a potential error reported by the analysis, because the context of all method calls (a client code) is arbitrary. In the bounded stack example, real errors were detected by a fabricated execution.

```

public class BoundedStack {
    private int[] elems;
    private int size;
    private int max;
    ...
    public BoundedStack(int capacity) {
        size = 0;
        // fixme: if (capacity <= 0) capacity = 2;
        max = capacity;
        elems=new int[max];
    }
    public void pop() {
        // fixme: if (size >= 0)
        size--;
    }
    public void push(int k) {
        int index;
        bool alreadyMember;
        alreadyMember = false;
        for(index=0; index<size; index++) {
            if(k==elems[index]) {
                alreadyMember = true;
                break;
            }
        }
        if (alreadyMember) {
            for (int j=index; j<size-1; j++) {
                elems[j] = elems[j+1];
            }
            elems[size-1] = k;
        }
        else {
            if (size < max) {
L1:    elems[size] = k;
                size++;
                return;
            } else {
                return;
            }
        }
    }
}

```

Figure 2.5: Implementation of a bounded stack using fixed-size array (abbreviated). The comments show the code needed to fix the errors.

procedure	fabricated states	abstract states	maximal length	description
search	2	21	5	searches a list for an element with a specified value
reverse	4	57	6	reverses a singly-linked list in-situ
insert	3	58	6	inserts a specified value into an ordered list
getLast	3	36	6	returns a pointer to the last element of a list
deleteAll	1	14	4	deallocates all elements in a list
delete	8	110	7	deletes an element with a specified value from a list

Figure 2.6: Analysis results for methods that manipulate singly-linked lists.

Recall that the purpose of fabrication is to find a proof faster. Inherent to this approach is the fact that a fabricated state may be unreachable from any initial state of the program. However, states that are fabricated on a method entry can be used in unit-test generation.

2.5.2 Prototype Implementation Based on TVLA with Application to Shape Analysis

Our method is applicable beyond predicate abstraction. We have implemented another prototype, based on the TVLA system [LAS00]. The TVLA system performs abstract interpretation using canonical abstraction [SRW02], and supports reasoning about linked data-structures. We have implemented a special-purpose model generator. For concrete execution, we use the TVLA system in a mode where memory abstraction is disabled. This mode faithfully simulates concrete state-space exploration for programs, which manipulate fields and pointers, but not integer data.

As a proof of concept, we applied the prototype to TVLA benchmarks that manipulate singly-linked lists. A concrete state describes a memory that contains linked lists. We used four test inputs, each with one linked list in memory: an empty list, and lists of length 1–3. The analysis proved the absence of null-dereferences and absence of memory leaks (i.e., every allocated element is reachable from some program variable). Fig. 2.6 shows the results: number of fabricated states, abstract states upon termination, and the maximal length of a list used by a concrete execution (either initial test input, or a fabricated state).

In fact, our prototype can find invariants for these methods without fabricating any states, provided that the input tests include a list with at least 7 nodes.⁴ It means that testing these methods on small lists, followed by abstraction, is sufficient to discover all reachable abstract states. In particular, every abstract state is weakly-reachable [Bal04] (see also Section 2.7).

2.6 Avoiding Unnecessary Abstraction Refinement

We are not the first to demonstrate that concrete execution plus abstraction can be used to verify program properties [LY92, PPV05, Bal04]. However, previous work in the area required much stronger abstractions than necessary to verify the safety properties of interest.

One approach is to find an abstract system that is *bisimilar* to the underlying concrete system, using automated refinement of abstractions [LY92, PPV05]. For deterministic systems, this means that the concrete system and abstract system have identical execution traces. The advantage of [LY92, PPV05], compared to our approach, is that it reports only real errors. However, this technique is often too strong for proving safety properties. Even for proving a simple program, bisimulation might require a complex abstraction generated via many iterations of abstraction-refinement, whereas our technique can achieve proofs with coarser abstractions.

As an example, we apply our method to the Bakery mutual exclusion protocol for two processes, which also

⁴The length depends on the number of program variables that can point into the same list.

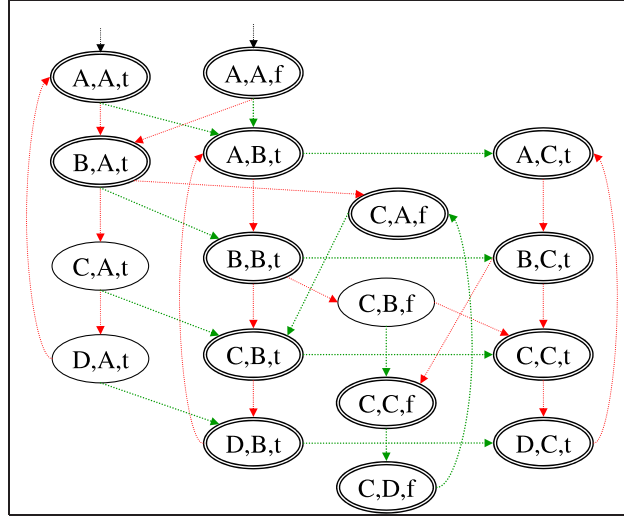


Figure 2.7: Reachable abstract states of two-process Bakery protocol, using an abstraction function $x_1 \leq x_2$.

was analyzed in [PPV05]. The guarded command representation of the protocol is:

Process1 :
 $pc_1 = A \quad \mapsto \quad x_1 := x_2; pc_1 = B;$
 $pc_1 = B \quad \mapsto \quad x_1 := x_1 + 1; pc_1 = C;$
 $pc_1 = C \wedge x_1 \leq x_2 \quad \mapsto \quad pc_1 = D;$
 $pc_1 = D \quad \mapsto \quad pc_1 = A;$

Process2 :
 $pc_2 = A \quad \mapsto \quad x_2 := x_1; pc_2 = B;$
 $pc_2 = B \quad \mapsto \quad x_2 := x_2 + 1; pc_2 = C;$
 $pc_2 = C \wedge x_2 < x_1 \quad \mapsto \quad pc_2 = D;$
 $pc_2 = D \quad \mapsto \quad pc_2 = A;$

A concrete state of the program is (pc_1, pc_2, x_1, x_2) , where pc_i is the value of the program counter of process i , ranging over $A - D$, and x_i is the integer value of the ticket of process i , for $i = 1, 2$.

The safety property we check is that at most one processor can be in the critical section: $\neg(pc_1 = D \wedge pc_2 = D)$. We use the following abstraction function, based on the predicate $x_1 \leq x_2$:

$$\alpha_{bake}(C) = \{(pc_1, pc_2, x_1 \leq x_2) \mid (pc_1, pc_2, x_1, x_2) \in C\} \quad (2.9)$$

Fig. 2.7 shows the reachable abstract states. The abstract error states (D, D, t) and (D, D, f) are not reachable. An explicit-state model checker easily finds a concrete execution that covers the states marked with bold-circles, starting from the concrete state $(A, A, 0, 0)$. Then, our method fabricates 2 states: $(C, B, 1, 0)$ and $(C, A, 0, 0)$. The first state covers the abstract state (C, B, f) , and the second state covers (C, A, t) and (D, A, t) . At this point, our method proves that the abstract states are invariant, and that they satisfy the mutual exclusion property.

The initial abstraction used in [PPV05] is the same as ours (2.9). The method of [PPV05] takes 4 steps of abstraction refinement to find an abstract state space that is a bisimulation of the concrete state space. The bisimilar abstract state space contains 36 abstract states, using abstraction based on 10 predicates. We have shown that the initial abstraction is sufficient to prove mutual exclusion (without any abstraction refinement), and the state space has only 17 abstract states.

2.7 Related Work

Automated Construction of Abstract Transformers Theorem provers have been used for the automated construction of abstract transition systems [BMMR01, HJMS03, YRS04, RSY04], especially in parametric abstract

domains, such as predicate abstraction [GS97] and canonical abstraction [SRW02], where the abstraction is defined per-program. In many cases, an exponential number of theorem prover calls is needed to compute the effect of a single program statement on an abstract value in the most-precise way.

Compared to these techniques, our method can reduce the number of theorem prover calls: it obtains abstract values via concrete execution, which does not require theorem-prover calls. A theorem prover is used only to check that an abstract value is an invariant (2.7). If the check fails, then at least one new abstract state is covered in the next iteration. In the worst case, our method might require as many theorem prover calls as other methods. However, if an execution from a fabricated state covers several new abstract states, our method terminates with fewer theorem prover calls.

The cost of a theorem prover call made by our method is comparable to other methods. However, the cost of a model generation might be higher than the cost of a validity check.

Our method is most-closely related to the algorithm presented in [RSY04]. Both methods rely on a model generator to “fabricate” a concrete state that (i) is not yet represented by the abstract value obtained so far, and (ii) is reachable in a single step from it. In this work, we have identified a way to cover more abstract states using a single fabricated state, by executing the program. The method of [RSY04] can be described by replacing $C := \text{execute}(f, T)$ with $C := T$ in line [5] of Fig. 2.3.

Bisimulation and Weak Reachability Concrete execution and abstraction are used by [LY92, PPV05, Bal04] to find errors and verify program properties. All errors reported by [LY92, PPV05] are real errors, but the technique often is too strong for proving safety, as shown in Section 2.6. Also in [PPV05], concrete exploration stops when it encounters a concrete state whose abstraction was already seen before. Our method continues exploration from such a concrete state, and may discover abstract states that were not covered before.

Another way to achieve verification is to find an abstraction and a set of tests T that cover exactly the reachable abstract states [Bal04]. It requires that every abstract state be testable. This is a weaker property than bisimulation but still stronger than our method, because our method using fabricated states may cover abstract states that are not reachable (but required for a proof).

Combining Dynamic and Static Analyses Daikon uses dynamic analysis to detect likely invariants [ECGN01]. It executes the program on a test set, examining the values of the concrete states, and detects patterns and relationships among those values. It reports properties that hold over execution of the given test set, but not necessarily over all program executions. In [NE02], likely invariants produced by Daikon are used with ESC/Java [LNS00] verification condition generator and Simplify [DNS03] theorem prover to prove that these are indeed invariants. Our work is similar in spirit, but uses fabricated states and abstraction to achieve proof via a fixed point computation where the Daikon-ESC/Java two-step process may fail to find a proof.

Recent work combines random test generation and concrete execution with symbolic execution and model generation [GKS05, SMA05, CS05]. These methods use symbolic techniques to direct the generation of tests towards unexplored paths in order to find errors faster. However, these methods do not employ abstraction, and in general cannot find proofs in presence of loops, with the exception of [GHK+06].

The method in [GHK+06] uses concrete execution to guide abstraction refinement to the program location both for the purpose of finding errors and the purpose of eliminating false alarms. This process might not terminate. On the other hand, our algorithm always terminates, but it can report potential errors, which might be false alarms. It is possible to combine our method with an analysis for classifying potential errors into real errors and false alarms, and use abstraction refinement to eliminate the latter. Our method uses concrete execution to find a proof faster, for a given abstract domain, without refining it. Both methods use a theorem prover to compute abstract transformers, but [GHK+06] does not use fabricated states to speed up the proof search. Similarly to our method, [GHK+06] can terminate with a proof which is a simulation and not necessarily a bisimulation. Our algorithm, unlike [GHK+06], does not require finite partitioning abstraction.

Adequacy Criterion for Testing We can define an *abstraction-based adequacy criterion* for a set of tests as follows. A set of tests T is adequate under a given abstraction if all reachable abstract states are covered by T . Note that if a set of tests T is adequate then safety properties can be (conservatively) checked on A_T —the abstract states covered by T . Our method checks a condition of invariance that implies adequacy.

In contrast to the traditional white-box adequacy criteria (e.g., [ZHM97]), we choose an abstraction based on the property of interest, and then define adequacy with respect to the abstraction. When used with a powerset abstraction, our adequacy requirement appears to be a formalization of partition-based testing with respect to an abstraction function, where each abstract state represents a partition.

Recently, an abstraction-based adequacy criteria *All-Abstract-States* was introduced in [Ere04], in the context of automatic test generation using a theorem prover, when the abstract states are provided by static analysis. All-Abstract-States criterion implies the adequacy criterion we defined in Section 2.1. Our algorithm provides an effective way to check adequacy of a given test set.

To summarize, our method can be viewed as bridging the gap [Har00, Ern03] between testing and verification. Our method complements existing techniques that combine dynamic and static analysis in that it is oriented towards finding a proof rather than detecting real errors.

Chapter 3

Computing Most-Precise Abstract Operations for Shape Analysis

The material described in this chapter is largely based on the material originally published in [YRS04]. In addition to the material already published in [YRS04], Appendix B contains a formal proof of correctness of the algorithm.

Shape analysis concerns the problem of determining “shape invariants” for programs that perform destructive updating on dynamically allocated storage. The motivation of the work presented in this chapter is to improve the precision, scalability and automation of shape analysis by employing theorem provers. This chapter presents a new algorithm that solves several open problems in shape analysis:

- Computing the most-precise abstract value that represents the (potentially infinite) set of states specified by a formula. We call this operation $\hat{\alpha}$.
- Computing the operation $assume[\varphi](a)$, which returns the most-precise abstraction of the set of states that are represented by a and satisfy a precondition φ .
- Implementing *best abstract transformers* for atomic program statements and conditions [CC79], as well as for loop-free code fragments (i.e., blocks of atomic program statements and conditions).
- Performing interprocedural shape analysis using procedure specifications and assume-guarantee reasoning.
- Computing the most-precise overapproximation of the meet of two abstract values.

In [Yor03, YRSW07], we show that the concretization of an abstract value for canonical abstraction can be expressed using a logical formula. Specifically, [Yor03, YRSW07] gives an algorithm that converts an abstract value a into a formula $\hat{\gamma}(a)$ that exactly characterizes $\gamma(a)$ —i.e., the set of concrete states that a represents.

As mentioned in Section 1.3.2, having $\hat{\gamma}$ and an algorithm for either $assume$ or $\hat{\alpha}$, we can implement all other operations mentioned above (see Fig. 1.3). To simplify the presentation of examples in this chapter, we describe the direct algorithm for $assume$ and demonstrate it using a running example throughout the chapter. We also give an algorithm for $\hat{\alpha}$, which is a simple variation of the algorithm for $assume$.

The algorithm employs a theorem prover for the logic used to express properties of data-structures. Candidate decidable logics for expressing such properties are described in [IRR⁺04a, YRS⁺06]. The algorithm can also be used with an undecidable logic and a theorem prover; termination can be assured by using standard techniques (e.g., having the theorem prover return a safe answer if a time-out threshold is exceeded) at the cost of losing the ability to guarantee that a most-precise result is obtained.

Prototype Implementation

To study the feasibility of our method, we have implemented a prototype of the $assume$ algorithm as an extension of the TVLA system [LAS00]. Our prototype uses the first-order theorem prover SPASS [Wei]. To perform

Relation	Intended Meaning
$x(v)$	Does pointer variable x point to element v ?
$y(v)$	Does pointer variable y point to element v ?
$n(v_1, v_2)$	Does the n field of v_1 point to v_2 ?
$eq(v_1, v_2)$	Do v_1 and v_2 denote the same element?
$is(v)$	Is v pointed to by more than one field ?

Figure 3.1: The set of relations for representing the states manipulated by programs that use the `List` data-type from Fig. 3.2 and two pointer variables x, y .

reasoning about (absence of) reachability in SPASS, we can provide, in some cases, sufficient first-order axiomatization of transitive closure, e.g., [LAIR⁺05].¹ So far, we tried three simple examples: two cases of *assume*, one of which is the running example of this chapter, and one case of best transformer. On all queries posed by these examples, the theorem prover terminated. The number of calls to SPASS in the running example is 158, and the overall running time was approximately 27 seconds.

The remainder of this chapter is organized as follows. In Section 3.1, we provide a short overview of canonical abstraction. The formal description of the *assume* and $\hat{\alpha}$ algorithms appears in Section 3.2, and the proof of correctness of *assume* algorithm is given in Appendix A. The algorithm that implements best abstract transformers is given in Section 3.3. In Section 3.4 we discuss related work.

3.1 Overview of Canonical Abstraction

This section provides a short overview of canonical abstraction. The formal description of the *assume* algorithm appears in Section 3.2.

As an example, consider the following precondition, expressed in C notation as: $(x \rightarrow n == y) \ \&\& \ (y \neq \text{null})$ (which will be abbreviated in this section as p), where x and y are program variables of the linked-list data-type defined in Fig. 3.2. The precondition p can be defined by a closed formula in first-order logic: $\varphi_0 \stackrel{\text{def}}{=} \exists v_1, v_2 : x(v_1) \wedge n(v_1, v_2) \wedge y(v_2)$. The operation $assume[p](a)$ enforces precondition p on an abstract value a . Typically, a represents a set of concrete states that may arise at the program point in which p is evaluated. The abstract value a used in the running example is depicted by the graph in Fig. 3.3(S). This graph is an abstraction of all concrete states that contain a non-empty linked list pointed to by x , as explained below.

3.1.1 3-Valued Structures

In this chapter, abstract values that are used to represent concrete states are sets of 3-valued logical structures over a vocabulary \mathcal{P} of predicate symbols. Each structure has a universe U of individuals and a mapping ι from k -tuples of individuals in U to values 1, 0, or 1/2 for each k -ary relation in \mathcal{P} . We say that the values 0 and 1 are **definite values** and that 1/2 is an **indefinite value**, meaning “either 0 or 1 possible”; a value l_1 is **consistent** with l_2 (denoted by $l_1 \sqsubseteq l_2$) when $l_1 = l_2$ or $l_2 = 1/2$; $\bigsqcup W$ denotes the least upper bound of the values in the set W .

A 3-valued structure provides a representation of states: individuals are abstractions of heap-allocated objects; unary relations represent pointer variables that point from the stack into the heap; binary relations represent pointer-valued fields of data-structures; and additional relations in \mathcal{P} describe certain properties of the heap. For example, a unary relation *is* (“is heap shared”) captures objects that are pointed to by more than one field. A special binary relation *eq* has the intended meaning of equality between locations. When the value of *eq* is 1/2 on the pair $\langle u, u \rangle$ for some node u , then u is called a “summary” node and it may represent more than one linked-list element. Fig. 3.1 describes the relations required for a program with pointer variables x and y , that manipulates the linked-list data-type defined in Fig. 3.2. 3-valued structures are depicted as directed graphs, with individuals as graph nodes. A relation with value 1 is represented by a solid arrow; with value 1/2 by a dotted arrow; and with value 0 by the absence of an arrow.

¹In general, there cannot be a complete, first-order axiomatization of transitive closure [Avr03, LAIR⁺05].

```

/* list.h */
typedef struct node {
    struct node *n;
    int data;
} *List;

```

Figure 3.2: A declaration of a linked-list data-type in C.

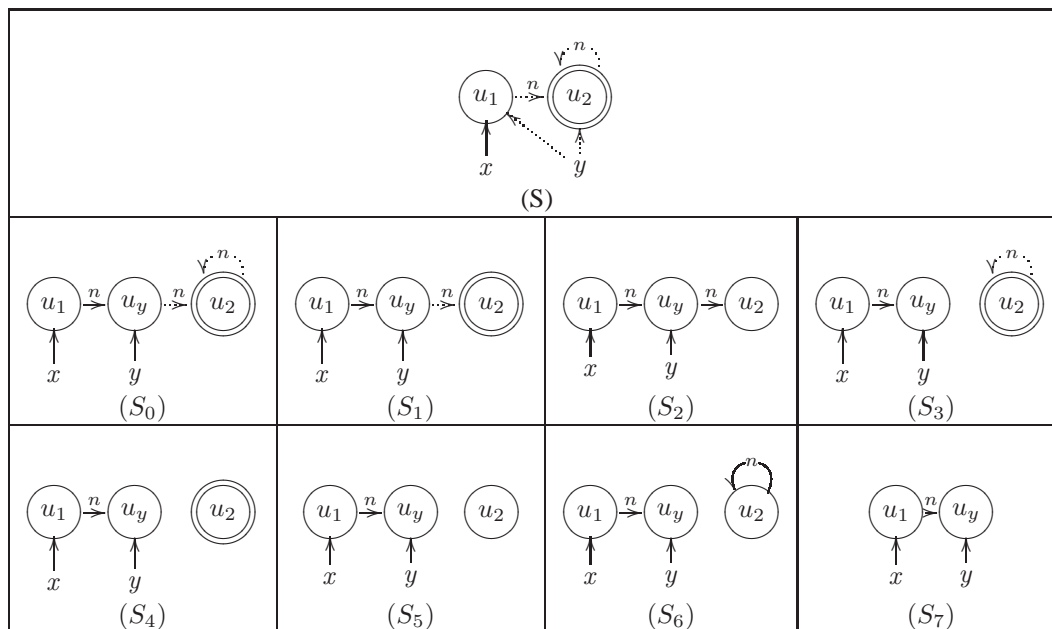


Figure 3.3: (S) The input abstract value $a = \{S\}$ represents all concrete states that contain a non-empty linked list pointed to by the program variable x , where the program variable y may point to some element. (S_0 – S_7) The result of computing $\text{assume}[p](a)$: the abstract value $a' = \{S_0, \dots, S_7\}$ represents all concrete states that contain a linked-list of length 2 or more that is pointed to by x , in which the second element is pointed to by y .

In Fig. 3.3(S), the solid arrow from x to the node u_1 indicates that relation x has the value 1 for the individual u_1 in the 3-valued structure S . This means that any concrete state represented by S contains a linked-list element pointed to by program variable x . Moreover, it **must** contain additional elements (represented by the summary node u_2 , drawn as a dotted circle), some of which **may** be reachable from the head of the linked-list (as indicated by the dotted arrow from u_1 to u_2 , which corresponds to the value $1/2$ of relation $n(u_1, u_2)$), and some of which **may** be linked to others (as indicated by the dotted self-arrow on u_2). The dotted arrows from y to u_1 and u_2 indicate that program variable y **may** point to any linked-list element. The absence of an arrow from u_2 to u_1 means that there is **no** n -pointer to the head of the list. Also, the unary relation is is 0 on all nodes and thus not shown in the graph, indicating that every element of a concrete state represented by this structure may be pointed to by at most one n -field.

We next introduce the subclass of bounded structures [SRW99]. Towards this end, we define **abstraction predicates** to be a designated subset of unary relations, denoted by \mathcal{Abs} . In the running example, all unary relations are defined as abstraction predicates. A **bounded structure** is a 3-valued structure in which for every pair of distinct nodes u_1, u_2 , there exists an abstraction predicate q such that q evaluates to distinct definite values for u_1 and u_2 . All 3-valued structures used throughout this chapter are bounded structures. Bounded structures are used in shape analysis to guarantee that the analysis is carried out w.r.t. a finite set of abstract structures, and hence will always terminate.

3.1.2 Embedding Order on 3-Valued Structures

3-valued structures are ordered by the **embedding order** (\sqsubseteq), defined below. $S \sqsubseteq S'$ guarantees that the set of concrete states represented by S is a subset of those represented by S' .

Let S and S' be two 3-valued structures, and let f be a surjective function that maps nodes of S onto nodes of S' . We say that f **embeds** S in S' (denoted by $S \sqsubseteq_f S'$) if for every relation $q \in \mathcal{P}$ of arity k and all k -tuples $\langle u_1, \dots, u_k \rangle$ in S , the value of q over $\langle u_1, \dots, u_k \rangle$ is consistent with, but may be more specific than, the value of q over $\langle f(u_1), \dots, f(u_k) \rangle$: $\iota^S(q)(u_1, \dots, u_k) \sqsubseteq \iota^{S'}(q)(f(u_1), \dots, f(u_k))$. We say that S **can be embedded into** S' (denoted by $S \sqsubseteq S'$) if there exists a function f such that $S \sqsubseteq_f S'$.

The result of $assume[p](a)$, shown in Fig. 3.3(S_0 – S_7), consists of 8 structures, each of which can be embedded into the input structure Fig. 3.3(S). The embedding function maps u_1 in each of the output structures S_0 – S_7 to the same node u_1 in the input structure. Each one of the output structures S_0 – S_6 contains nodes u_y and u_2 , both of which are mapped by the embedding to u_2 in S ; for S_7 , node u_y is mapped to u_2 in S . Thus, concrete elements represented by different nodes u_y and u_2 in the output structures are represented by a single summary node u_2 in the input structure. We say that node u_y is “materialized” from node u_2 . As we shall see, this is the only new node required to guarantee the most-precise result, relative to the abstraction.

For each of S_0, \dots, S_7 , the embedding function described above is consistent with the values of the relations. The value of x on u_1 is 1 in S_i and S structures. Indefinite values of relations in S impose no restriction on the corresponding values in the output structures. For instance, the value of y is $1/2$ on all nodes in S , which is consistent with its value 0 on nodes u_1 and u_2 and the value 1 on u_y in each of S_0, \dots, S_7 . The absence of an n -edge from u_2 back to u_1 in S implies that there must be no edge from u_y to u_1 and from u_2 to u_1 in the output structures, i.e., the values of the relation n on these pairs must be 0.

3.1.3 Integrity Rules

A 2-valued structure is a special case of a 3-valued structure, in which relation values are only 0 and 1. Because not all 2-valued structures represent valid concrete states, we use a designated set of **integrity rules**, to exclude impossible states. The integrity rules are fixed for each particular analysis and defined by a conjunction of closed formulas over the vocabulary \mathcal{P} , that must be satisfied by all concrete states. For the linked-list data-type in Fig. 3.2, the following conditions define the valid concrete states: (i) each program variable can point to at most one heap node, (ii) the n -field of an element can point to at most one element, (iii) $is(v)$ holds if and only if there exist two distinct elements with n -fields pointing to v . Finally, eq is given the interpretation of equality: $eq(v_1, v_2)$ holds if and only if v_1 and v_2 denote the same element.

3.1.4 Canonical Abstraction

The abstraction we use throughout this chapter is **canonical abstraction**, as defined in [SRW02]. The surjective function β takes a 2-valued structure and returns a 3-valued structure with the following properties:

- β maps concrete nodes into abstract nodes according to **canonical names** of the nodes, constructed from the values of the abstraction predicates.
- β is a **tight embedding** [SRW02], i.e., the value of the relation q on an abstract node-tuple is $1/2$ only when there exist two corresponding concrete node-tuples with distinct values.

A 3-valued structure S is an ICA (Image of Canonical Abstraction) if there exists a 2-valued structure S^{\natural} such that $S = \beta(S^{\natural})$. Note that every ICA is a bounded structure.

For example, all structures in Fig. 3.3(S_0 – S_7) produced by $assume[p](a)$ operation are ICAs, whereas the structure in Fig. 3.3(S) is not an ICA. The structure in Fig. 3.3(S_1) is a canonical abstraction of the concrete structure in Fig. 3.4(a) and also the one in Fig. 3.4(b).

The abstraction function α is defined by extending β pointwise, i.e., $\alpha(W) = \{\beta(S^{\natural}) \mid S^{\natural} \in W\}$ where W is a set of 2-valued structures. The concretization function γ takes a set of 3-valued structures W and returns a potentially infinite set of 2-valued structures $\gamma(W)$ where $S^{\natural} \in \gamma(W)$ iff S^{\natural} satisfies the integrity rules and there exists $S \in W$ such that $\beta(S^{\natural}) \sqsubseteq S$.

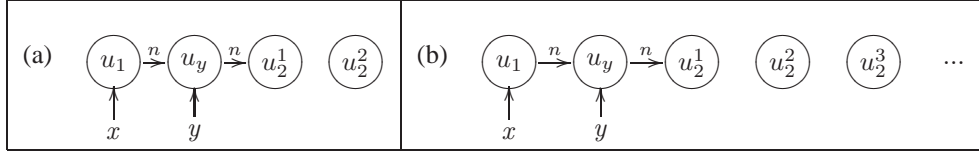


Figure 3.4: Concrete states represented by the structure S_1 from Fig. 3.3. (a) The concrete nodes u_2^1 and u_2^2 are mapped to the abstract node u_2 . (b) The concrete nodes u_2^1 , u_2^2 and u_2^3 are mapped to the abstract node u_2 . More concrete structures can be generated in the same manner, by adding more isolated nodes that map to the summary node u_2 .

The requirement of $assume[p](a)$ to produce the most-precise abstract value amounts to producing $\alpha(X)$, where X is the set of concrete structures that embed into a and satisfy p . Indeed, the result of $assume[p](a)$ in Fig. 3.3(S_0 – S_7) satisfies this requirement, because S_0 – S_7 are the canonical abstractions of all structures in X .

For example, structure S_1 from Fig. 3.3 is a canonical abstraction of each of the structures in Fig. 3.4. However, S_1 is not a canonical abstraction of S_2 from Fig. 3.3,² because the value $1/2$ of n for $\langle u_y, u_2 \rangle$ requires that a concrete structure abstracted by S_1 have two pairs of nodes with the same canonical names as $\langle u_y, u_2 \rangle$ and with distinct values of n . This requirement does not hold in S_2 , because it contains only one pair $\langle u_1, u_2 \rangle$ with those canonical names. Without S_2 , the result would not include the canonical abstractions of all concrete structures in X , but it would be semantically equivalent (because S_2 can be embedded into S_1). The version of the $assume[p](a)$ algorithm that we describe does include S_2 in the output. It is straightforward to generalize the algorithm to produce the smallest semantically equivalent set of structures.

It is non-trivial to produce the most-precise result for $assume[p](a)$. For instance, in each of S_0 – S_6 there is no back-edge from u_2 to u_y even though both nodes embed into the node u_2 of the input structure, which has a self-loop with n evaluating to $1/2$. It is a consequence of the integrity rules that no back-edge can exist from any u_2^j to u_y in any concrete structure that satisfies p : precondition p implies the existence of an n -pointer from u_1 to u_y , but u_y cannot have a second incoming n -edge (because the value of the relation is on u_y is 0).

Consequently, to determine relation values in the output structure, each concrete structure that it represents must be accounted for. Because the number of such concrete structures is potentially infinite, they cannot be examined explicitly. The algorithm described here uses a theorem prover to perform this task symbolically.

Towards this end, the algorithm uses a symbolic representation of concrete states as a logical formula, called a **characteristic formula**. The characteristic formula for an abstract value a is denoted by $\hat{\gamma}(a)$; it is satisfied by a 2-valued structure S^{\natural} if and only if $S^{\natural} \in \gamma(a)$. The $\hat{\gamma}$ formula for shape analysis is defined in [Yor03, YRSW07] for bounded structures, and it includes the integrity rules.

In addition, a necessary requirement for the output of $assume$ to be a set of ICAs is imposed by the formula $\varphi_{q, u_1, \dots, u_k}$, defined in (3.1) below; this is used to check whether the value of a relation q can be $1/2$ on a node-tuple $\langle u_1, \dots, u_k \rangle$ in a structure S . Intuitively, the formula is satisfiable when there exists a concrete structure represented by S that contains two tuples of nodes, both mapped to the abstract tuple $\langle u_1, \dots, u_k \rangle$, such that q evaluates to distinct values on these tuples. If the formula is not satisfiable, S is not a result of canonical abstraction, because the value of q on $\langle u_1, \dots, u_k \rangle$ is not as precise as possible, compared to the value of q on the corresponding concrete nodes.

3.2 The *assume* Algorithm

The *assume* algorithm is shown in Fig. 3.5. It takes a formula φ and a set of bounded structures a , and computes the set of ICA structures that are represented by a and satisfy φ .

The algorithm operates in two phases. Phase 1 of the algorithm performs node “materialization”: if a structure has an indefinite value of an abstraction predicate q on some abstract node, the node may be *bifurcated* into two nodes and q is set to distinct definite values on the new nodes. As a result of this phase, all the abstraction predicates have definite values. Phase 2 refines the structures produced in phase 1 by lowering relation values

² S_2 is a 2-valued structure, and is a canonical abstraction of itself.

from 1/2 to 0 and 1. Both phases use a theorem prover to filter out abstract structures that do not represent any 2-valued structures that satisfy φ .

Section 3.2.1 explains the role of the theorem prover and the queries posed by our algorithm. The algorithm is explained in Section 3.2.2 (phase 1) and Section 3.2.3 (phase 2). Finally, the properties of the algorithm are discussed in Section 3.2.4.

```

procedure assume( $\varphi$ : Formula,  $a$ : a set of bounded structures): Set of ICA structures
  result :=  $a$ 
  // Phase 1
  result := bif( $\varphi$ , result)
  // Phase 2
  while there exists  $S \in \textit{result}$ ,  $q \in \mathcal{P}$  of arity  $k$ , and  $u_1, \dots, u_k \in U^S$  such that
     $\iota^S(q)(u_1, \dots, u_k) = 1/2$  and  $\textit{done}(S, q, u_1, \dots, u_k) = \textit{false}$  do
       $\textit{done}(S, q, u_1, \dots, u_k) := \textit{true}$ 
      if not isSatisfiable( $\widehat{\gamma}(S) \wedge \varphi \wedge \varphi_{q, u_1, \dots, u_k}$ ) then result := result \ { $S$ }
       $S_0 := S[q(u_1, \dots, u_k) \mapsto 0]$ 
      if isSatisfiable( $\widehat{\gamma}(S_0) \wedge \varphi$ ) then result := result  $\cup$  { $S_0$ }
       $S_1 := S[q(u_1, \dots, u_k) \mapsto 1]$ 
      if isSatisfiable( $\widehat{\gamma}(S_1) \wedge \varphi$ ) then result := result  $\cup$  { $S_1$ }
  return result

```

Figure 3.5: The *assume* procedure takes a formula φ over the vocabulary \mathcal{P} and a set of bounded structures a , and computes the set of ICA structures *result*. The characteristic formula computed by $\widehat{\gamma}$ includes the integrity rules in order to eliminate infeasible concrete structures. The formula $\varphi_{q, u_1, \dots, u_k}$ is defined in (3.1). The procedure *bif*(φ , *result*) is shown in Fig. 3.6. The flag $\textit{done}(S, q, u_1, \dots, u_k)$ marks processed q -tuples; initially, *done* is *false* for all relation tuples.

3.2.1 Employing a Theorem Prover

The formula $\varphi_{q, u_1, \dots, u_k}$ guarantees that a concrete structure must contain two tuples of nodes, both mapped to the abstract tuple $\langle u_1, \dots, u_k \rangle$, on which q evaluates to distinct values. This is captured by the formula

$$\begin{aligned} \varphi_{q, u_1, \dots, u_k} \stackrel{\text{def}}{=} & \exists w_1^1, \dots, w_k^1, w_1^2, \dots, w_k^2 : \bigwedge_{i=1}^k \textit{node}_{u_i}^S(w_i^1) \wedge \bigwedge_{i=1}^k \textit{node}_{u_i}^S(w_i^2) \\ & \wedge \bigwedge_{i=1}^k \textit{eq}(w_i^1, w_i^2) \wedge q(w_1^1, \dots, w_k^1) \wedge \neg q(w_1^2, \dots, w_k^2) \end{aligned} \quad (3.1)$$

$\varphi_{q, u_1, \dots, u_k}$ uses the *node* formula, originally defined in [Yor03], which uniquely identifies the mapping of concrete nodes into abstract nodes. For a bounded structure S , $\textit{node}_u^S(v)$ simply asserts that u and v agree on all abstraction predicates.

The function *isSatisfiable*(ψ) invokes a theorem prover that returns *true* when ψ is satisfiable, i.e., the set of 2-valued structures that satisfy ψ is non-empty. This function guides the refinement of relation values. In particular, the satisfiability of a formula ψ is used to make the following decisions:

- Discard a 3-valued structure S that does not represent any concrete state that satisfies φ by taking $\psi \stackrel{\text{def}}{=} \widehat{\gamma}(S) \wedge \varphi$.
- Materialize a new node from node u w.r.t. the value of $q \in \mathcal{Abs}$ in S (phase 1) by taking $\psi \stackrel{\text{def}}{=} \widehat{\gamma}(S) \wedge \varphi \wedge \varphi_{q, u}$.
- Retain the indefinite value for relation q on node-tuple $\langle u_1, \dots, u_k \rangle$ in S (in phase 2) by taking $\psi \stackrel{\text{def}}{=} \widehat{\gamma}(S) \wedge \varphi \wedge \varphi_{q, u_1, \dots, u_k}$.

This requires a theorem prover for the logic that expresses φ , $\varphi_{q, u}$ and $\widehat{\gamma}$, including the integrity rules.

```

procedure bif( $\varphi$ : Formula,  $W$ : Set of bounded structures): Set of bounded structures
for all  $S \in W$ 
  if not isSatisfiable( $\widehat{\gamma}(S) \wedge \varphi$ ) then  $W := W \setminus \{S\}$ 
while there exists  $S \in W, q \in \mathcal{Abs}$  and  $u \in U^S$  such that  $\iota^S(q)(u) = 1/2$ 
   $W := W \setminus \{S\}$ 
  if isSatisfiable( $\widehat{\gamma}(S) \wedge \varphi \wedge \varphi_{q,u}$ ) then  $W := W \cup S[u \mapsto u.0, u.1][q(u.0) \mapsto 0, q(u.1) \mapsto 1]$ 
   $S_0 := S[q(u) \mapsto 0]$ 
  if isSatisfiable( $\widehat{\gamma}(S_0) \wedge \varphi$ ) then  $W := W \cup \{S_0\}$ 
   $S_1 := S[q(u) \mapsto 1]$ 
  if isSatisfiable( $\widehat{\gamma}(S_1) \wedge \varphi$ ) then  $W := W \cup \{S_1\}$ 
return  $W$ 

```

Figure 3.6: The procedure takes a set of structures and a formula φ over the vocabulary \mathcal{P} , and computes the bifurcation of each structure in the input set, w.r.t. the input formula. Note that at the beginning of the procedure, it ensures that each structure in the working set W represents at least one concrete structure that satisfies φ . The formula $\varphi_{q,u}$ is defined in (3.1). The operation $S[u \mapsto u.0, u.1]$ performs a bifurcation of the node u in S , setting the values of all relations on $u.0$ and $u.1$ to the values they had on u .

3.2.2 Materialization

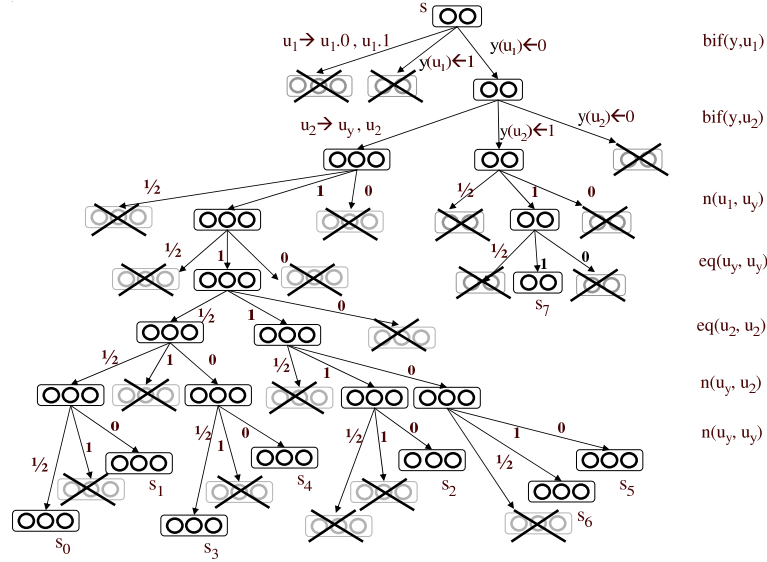
Phase 1 of the algorithm performs node “materialization” by invoking the procedure *bif*. The name *bif* comes from its main purpose: whenever a structure has an indefinite value of an abstraction predicate q on some abstract node, supported by distinct values on corresponding concrete nodes, the node is *bifurcated* into two nodes and q is set to distinct definite values on the new nodes. The *bif* procedure produces a set of 3-valued structures that have the same set of canonical names as the concrete states that satisfy φ and embed into a . The *bif* procedure first filters out potentially unsatisfiable structures, and then iterates over all structures $S \in W$ that have an indefinite value for an abstraction predicate $q \in \mathcal{Abs}$ on some node u . It replaces S by other structures. As a result of this phase, all the abstraction predicates have definite values for all nodes in each of the structures. Because the output structures are bounded structures, the number of different structures that can be produced is finite, which guarantees that *bif* procedure terminates.

In the body of the loop in *bif*, we check if there exists a concrete structure represented by S that satisfies φ in which q has distinct values on concrete nodes represented by u (the query is performed using the formula $\varphi_{q,u}$). In this case, a new structure S' is added to W , created from S by duplicating the node u in S into two instances and setting the value of q to 0 for one node instance, and to 1 for another instance. All other relation values on the new node instances are the same as their values on u .

In addition, two copies of S are created with 0 and 1, respectively, for the value of $q(u)$. To guarantee that each copy represents a concrete structure that satisfies φ an appropriate query is posed to the theorem prover. Omitting this query will produce a sound, but potentially overly-conservative result.

Fig. 3.7 shows a computation tree for the algorithm on the running example. A node in the tree is labeled by a 3-valued structure, sketched by showing its nodes. Its children are labeled by the result of refining the 3-valued structure w.r.t. the relation and the node-tuple on the right, by the values shown on the outgoing edges.

The order in which relation values are examined affects the complexity (in terms of the number of calls to a theorem prover, the size of the query formulas in each call and the maximal number of explored structures), but it does not affect the result, provided that all calls terminate. The order in Fig. 3.7 was chosen for convenience of presentation. The root of the tree contains the sketch of the input structure S from Fig. 3.3(S); u_1 is the left circle and u_2 is the right circle. Fig. 3.7 shows the steps performed by *bif* on the input $\{S\}$ in Fig. 3.3. *bif* examines the abstraction predicate y , which has indefinite values on the nodes u_1 and u_2 . The algorithm attempts to replace S by T' , T_1 , and T_0 , shown as the children of S in Fig. 3.7. The structures T' and T_1 are discarded because all of the concrete structures they represent violate integrity rule (i) for x (Section 3.1.3) and the precondition p , respectively. The remaining structure T_0 is further modified w.r.t. the value of $y(u_2)$. However, setting $y(u_2)$ to 0 results in a structure that does not satisfy p , and hence it is discarded.

Figure 3.7: A computation tree for $assume[p](a)$ for a shown in Fig. 3.3.

3.2.3 Refining Relation Values

The second phase of the *assume* algorithm refines the structures by lowering relation values from $1/2$ to 0 and 1 , and throwing away a structure S when it has a relation q with the value $1/2$ for some tuple $q(u_1, \dots, u_k)$, but S does not represent any 2-valued structure with corresponding tuples $q(u'_1, \dots, u'_k) = 0$ and $q(u''_1, \dots, u''_k) = 1$.

For each structure S and an indefinite value of a relation $q \in \mathcal{P}$ on a tuple of abstract nodes, we eliminate structures in which the relation has the same values on all corresponding tuples in all concrete structures that are represented by S and satisfy φ . (This query is performed using the formula in (3.1).) In addition, two copies of S are created with the values 0 and 1 for q , respectively. To guarantee that each copy represents a concrete structure that satisfies φ , an appropriate query is posed to a theorem prover. The *done* flag is used to guarantee that each relation tuple is processed only once.

The bulk of Fig. 3.7 (everything below the top two rows) shows the refinement of each relation value in the running example. Phase 2 starts with two structures, T'_2 and T'_3 , of size 2 and 3, produced by *bif*. Consider the refinement of T'_2 w.r.t. $n(u_1, u_y)$, where u_1 is pointed to by x and u_y is pointed to by y (the same node names as in Fig. 3.3).

The relation tuple $n(u_1, u_y)$ cannot be set to $1/2$, because it requires the existence of a concrete structure with two different pairs of nodes mapped to $\langle u_1, u_y \rangle$; however, integrity rule (i) in Section 3.1.3 implies that there is exactly one node represented by u_1 and exactly one node represented by u_y . Intuitively, this stems from the fact that the (one) concrete node represented by $u_1(u_y)$ is pointed to by $x(y)$. The relation tuple $n(u_1, u_y)$ cannot be set to 0 , because this violates the precondition p , according to which the element pointed to by y (represented by u_y) must also be pointed to by the n -field of the element pointed to by x (represented by u_1). Guided by the computation tree in Fig. 3.7, the reader can verify that the structures in Fig. 3.3(S_0 – S_7) are generated by $assume[p](a)$. (The final answer is read out at the leaves).

3.2.4 Properties of the Algorithm

We determine the complexity of the algorithm in terms of (i) the size of each structure, i.e., the number of nodes and definite values, (ii) the number of structures, and (iii) the number of the calls to the theorem prover. The size of each query formula passed to the theorem prover is linear in the size of the examined structure, because $\widehat{\gamma}(S)$ is linear in S , φ is usually small, and the size of $\varphi_{q,u}$ is fixed for a given \mathcal{P} . The complexity in terms of (ii) and (iii) is linear in the height of the abstract domain of sets of ICA structures defined over \mathcal{P} . The abstract domain is doubly-exponential in the size of \mathcal{P} , and its height is exponential in the size of \mathcal{P} . Therefore, our algorithm is exponentially more efficient than the naïve **enumerate-and-eliminate** algorithm over the abstract domain.

Let X denote the set $\llbracket \varphi \rrbracket \cap \gamma(a)$. To prove the correctness of the algorithm, it is sufficient to establish the following properties (the proofs appear in Appendix B):

1. All the structures explored by the algorithm are bounded structures.
2. $result \sqsupseteq \alpha(\llbracket \varphi \rrbracket \cap \gamma(a))$. This requirement ensures that the result is **sound**, i.e., $result$ contains canonical abstractions of all concrete structures in X . This is a global invariant throughout the algorithm.
3. $result \sqsubseteq \alpha(\llbracket \varphi \rrbracket \cap \gamma(a))$. This requirement ensures that $result$ does not contain abstract structures that are not ICAs of any concrete state in X . This holds upon the termination of the algorithm.

3.2.5 Computing $\hat{\alpha}$

As mentioned in Section 1.3.2, we can implement $\hat{\alpha}(\varphi)$ by $assume[\varphi](\top)$, where \top denotes the abstract value that represents all possible concrete states (the largest value in the abstract domain). A direct algorithm for $assume$ is a slight modification of the algorithm in Fig. 3.5, in which we replace the first line by $result := \top$.

3.3 Implementing the Best Transformer

We can use the $assume$ operation to implement the best transformer for canonical abstraction. More specifically, we can use $assume$ to compute the result of the best transformer.

The best-transformer algorithm manipulates the two-store vocabulary $\mathcal{P} \cup \mathcal{P}'$, which includes two copies of each relation — the original unprimed one, as well as a primed version of the relation. The original version of the relation contains the values before the transformer is applied, and the primed version contains the new values.

The best-transformer algorithm takes a set of bounded structures a over a vocabulary \mathcal{P} , and a transformer formula τ over the two-store vocabulary $\mathcal{P} \cup \mathcal{P}'$. It returns a set of ICA structures over the two-store vocabulary that is the canonical abstraction of all pairs of concrete structures $\langle S_1^h, S_2^h \rangle$ such that S_2^h is the result of applying the transformer τ to S_1^h . $BT(\tau, a)$ is computed by $assume(\tau, extend(a))$ that operates over the two-store vocabulary, where $extend(a)$ extends each structure in $S \in a$ into one over a two-store vocabulary by setting the values of all primed relations to $1/2$. The result of the best transformer can be obtained from the primed version of the relations in the output structure.

The two-store vocabulary allows us to maintain the relationship between the values of the relations before and after the transformer. Also, τ is an arbitrary formula over the two-store vocabulary; in particular, it may contain a precondition that involves unprimed versions of the relations, together with primed relations in the “update” part.

3.4 Related Work

In [RSY04], we have presented a different technique to compute best transformers in a more general setting of finite-height, but possibly infinite-size lattices. The technique presented in [RSY04] handles infinite domains by requiring that a theorem prover produce a concrete counter-example for invalid formulas, which is not required for the algorithm presented in this chapter.

Compared to [RSY04], an advantage of the approach taken in this chapter is that it iterates from above: it always holds a legitimate value (although not the best). If the logic is undecidable, a timeout can be used to terminate the computation and return the current value. Because the technique described in [RSY04] starts from \perp , an intermediate result cannot be used as a safe approximation of the desired answer. Another potential advantage of the approach in this chapter is that the size of formulas in the algorithm reported here is linear in the size of structures (counting 0 and 1 values), and does not depend on the height of the domain.

This chapter is also closely related to past work on predicate abstraction, which also uses theorem provers to implement most-precise versions of the basic abstract-interpretation operations. Predicate abstraction is a special case of canonical abstraction, when only nullary relations are used. Interestingly, when applied to a vocabulary with only nullary relations, the algorithm in Fig. 3.5 is similar to the algorithm used in SLAM [BR01]. It starts with $1/2$ for all of the nullary relations and then repeatedly refines instances of $1/2$ into 0 and 1. The more general setting of canonical abstraction requires us to use the formula $\varphi_{q, u_1, u_2, \dots, u_k}$ to identify the appropriate values of

non-nullary relations. Also, we need the first phase (procedure *bif*) to identify what node materializations need to be carried out.

The algorithm described in this chapter was inspired by the Focus operation in TVLA, which is similar in spirit to the *assume* operation. The input of Focus is a set of 3-valued structures and a formula φ . Focus returns a semantically equivalent set of 3-valued structures in which φ evaluates to a definite value, according to the Kleene semantics for 3-valued logic [SRW02]. The *assume* algorithm reported in this chapter has the following advantages. First, it guarantees that the number of resultant structures is finite. The Focus algorithm in TVLA generates a runtime exception when this cannot be achieved. This makes Focus a partial function, which was sometimes criticized by the TVLA user community. Second, the number of structures generated by *assume* is optimal in the sense that it never returns a 3-valued structure unless it is the canonical abstraction of some required state.

The latter property is achieved using an off-the-shelf theorem prover; which makes *assume* currently slower than Focus. To enjoy the benefits of *assume* while maintaining efficiency, it is possible to develop a specialized theorem prover, as the one discussed in the next chapter. It is also possible to reduce the number of theorem prover calls made by *assume*. For example, we can avoid an expensive theorem prover call if Kleene evaluation of the formula returns a definite value.

Perhaps the most exciting future application of *assume* is for modular analysis with assume-guarantee reasoning. It would permit TVLA to be applied to large programs by using procedure specifications. The challenge would be to identify an expressive specification language and to implement a fast decision procedure for reasoning about these specifications.

To summarize, for shape-analysis problems, the methods described in this chapter are more automatic and more precise than the ones used in TVLA, and allow modular analysis with assume-guarantee reasoning, although they are currently much slower.

Chapter 4

Logic of Reachable Patterns in Linked Data-Structures

We define a new decidable logic for expressing and checking invariants of programs that manipulate dynamically-allocated objects via pointers and destructive pointer updates. The main feature of this logic is the ability to limit the neighborhood of a node that is reachable via a regular expression from a designated node. The logic is closed under boolean operations (entailment, negation) and has a finite model property. The key technical result is the proof of decidability.

We show how to express preconditions, postconditions, and loop invariants for some interesting programs. It is also possible to express properties such as disjointness of data-structures, and low-level heap mutations. Moreover, our logic can express properties of arbitrary data-structures and of an arbitrary number of pointer fields. The latter provides a way to naturally specify postconditions that relate the fields on the entry of a procedure to the field on the exit of a procedure. Therefore, it is possible to use the logic to automatically prove partial correctness of programs performing low-level heap mutations.

The material described in this chapter was originally published in [YRS⁺06], and an extended version of [YRS⁺06] was invited for a journal publication and appeared in [YRS⁺07]. In addition to the material already published in [YRS⁺06, YRS⁺07], Section 4.3.4 shows that *LRP* can be used to characterize certain shape abstractions, and Section 4.7.2 contains a proof of the upper bound on the complexity of checking satisfiability of *LRP* formulas.

This chapter is organized as follows: Section 4.1 defines the syntax and the semantics of \mathcal{L}_0 , and shows that it has a finite model property; Section 4.2 shows that \mathcal{L}_0 is undecidable; Section 4.3 defines the fragment \mathcal{L}_1 , and demonstrates the expressiveness of \mathcal{L}_1 on several examples. Section 4.4 presents the decidability proof for \mathcal{L}_1 , with a detailed proof of the main theorem given in Section 4.5; Section 4.6 defines an interesting extension of \mathcal{L}_1 , called \mathcal{L}_2 , and sketches the proof of decidability of \mathcal{L}_2 , which does not immediately follow from that of \mathcal{L}_1 ; Section 4.7 contains the complexity results for \mathcal{L}_1 ; Section 4.8 discusses the limitations and the extensions of the new logics; finally, Section 4.9 discusses related work.

4.1 The \mathcal{L}_0 Logic

In this section, we define the syntax and the semantics of our logic. For simplicity, we explain the material in terms of expressing properties of heaps. However, our logic can actually model properties of arbitrary directed graphs. Still, the logic is powerful enough to express the property that a graph denotes a heap.

4.1.1 Syntax of \mathcal{L}_0

\mathcal{L}_0 is a propositional logic over reachability constraints. That is, an \mathcal{L}_0 formula is a boolean combination of closed formulas in first-order logic with transitive closure that satisfy certain syntactic restrictions.

Let $\tau = \langle C, U, F \rangle$ denote a vocabulary, where

- C is a finite set of constant symbols usually denoting designated objects in the heap, pointed to by program variables;
- U is a set of unary relation symbols denoting properties, e.g., the color of a node in a Red-Black tree;
- F is a finite set of binary relation symbols (edges) usually denoting pointer fields.¹

For example, we can describe a doubly-linked list with forward pointer f and backward pointer b , pointed-to by a program variable x , using the vocabulary in which $C = \{x\}$, $U = \{\}$, and $F = \{f, b\}$. We can describe a tree pointed-to by the program variable $root$, in which each node contains a data value from a finite set of values D , using the vocabulary in which $C = \{root\}$, $F = \{r, l\}$, and U contains a symbol for each value of D .

A **term** t is either a variable or a constant. An **atomic formula** is an equality $t = t'$, a monadic formula $u(t)$ for some $u \in U$, or an edge formula $t \xrightarrow{f} t'$ for some $f \in F$, and terms t, t' . A **quantifier-free formula** $\psi(v_0, \dots, v_n)$ over τ and variables v_0, \dots, v_n is an arbitrary boolean combination of atomic formulas. We say that a sub-formula ψ appears positively (negatively) in φ , if ψ appears under an even (odd) number of negations in φ . Let $FV(\psi)$ denote the free variables of the formula ψ .

Definition 4.1.1 A **neighborhood formula** $N(v_0, \dots, v_n)$ is a conjunction of edge formulas of the form $v \xrightarrow{f} v'$, where $f \in F$ and $v, v' \in \{v_0, \dots, v_n\}$, and monadic formulas of the form $u(v)$ or $\neg u(v)$, where $u \in U$.

Definition 4.1.2 Let $N(v_0, \dots, v_n)$ be a neighborhood formula. The **Gaifman graph** of N , denoted by B_N , is an undirected graph with a vertex for each free variable of N . There is an edge between the vertices corresponding to v_i and v_j in B_N if and only if $(v_i \xrightarrow{f} v_j)$ or $(v_j \xrightarrow{f} v_i)$ appears in N , for some $f \in F$. The **distance** between logical variables v_i and v_j in the formula N is the minimal edge distance between the corresponding vertices v_i and v_j in B_N .

For example, for the formula $N = (v_0 \xrightarrow{f} v_1) \wedge (v_0 \xrightarrow{f} v_2)$ the distance between v_1 and v_2 in N is 2, and its underlying graph B_N looks like this: $v_1 - v_0 - v_2$.

Definition 4.1.3 A **routing expression** is an extended regular expression, defined as follows:

$R ::=$	\emptyset			<i>empty set</i>
	ϵ			<i>empty path</i>
	\xrightarrow{f}	$f \in F$		<i>forward along edge</i>
	\xleftarrow{f}	$f \in F$		<i>backward along edge</i>
	u	$u \in U$		<i>test if u holds</i>
	$\neg u$	$u \in U$		<i>test if u does not hold</i>
	c	$c \in C$		<i>test if c holds</i>
	$\neg c$	$c \in C$		<i>test if c does not hold</i>
	$R_1.R_2$			<i>concatenation</i>
	$R_1 R_2$			<i>union</i>
	R^*			<i>Kleene star</i>

Intuitively, a routing expression describes a path in the heap.

A routing expression can require that a path traverse some pointer fields backwards. For example, the routing expression $\xrightarrow{f}^* \cdot \xleftarrow{f}^*$ describes a sequence of f -edges that may look like this: $\xrightarrow{f} \xrightarrow{f} \xleftarrow{f} \xleftarrow{f} \xleftarrow{f}$. We use this routing expression in Section 4.3.2 to describe disjoint data-structures.

A routing expression has the ability to test properties of heap objects along the path. For example, a routing expression $(\xrightarrow{f} \cdot \neg y)^*$ describes a path which does not traverse an object pointed-to by the program variable y . We use this routing expression to describe a path along which some property holds *until* the path reaches the object pointed-to by y (see Section 4.3.2).

Definition 4.1.4 (Syntax of \mathcal{L}_0) A **reachability constraint** is a closed formula of the form:

$$\forall v_0, \dots, v_n. R(c, v_0) \Rightarrow (N(v_0, \dots, v_n) \Rightarrow \psi(v_0, \dots, v_n)) \quad (4.1)$$

¹We can also allow auxiliary constants and fields including abstract fields [BCC⁺05].

where $c \in C$ is a constant, R is a routing expression, N is a neighborhood formula, and ψ is an arbitrary quantifier-free formula, such that $FV(N) \subseteq \{v_0, \dots, v_n\}$ and $FV(\psi) \subseteq FV(N) \cup \{v_0\}$. In particular, if the neighborhood formula N is true (the empty conjunction), then ψ is a formula with a single free variable v_0 .

An \mathcal{L}_0 **formula** is a boolean combination of reachability constraints.

The subformula $R(c, v_0)$ defines an R -labelled path from c to v_0 . The subformula $N(v_0, \dots, v_n) \Rightarrow \psi(v_0, \dots, v_n)$ defines a **pattern**, denoted by $p(v_0)$. Here, the designated variable v_0 denotes the “central” node of the “neighborhood” reachable from c by following an R -path. Intuitively, neighborhood formula N binds the variables v_0, \dots, v_n to nodes that form a subgraph, and ψ defines more constraints on those nodes.²

For example, the pattern $det_f(v_0)$ defined by the formula $(v_0 \xrightarrow{f} v_1) \wedge (v_0 \xrightarrow{f} v_2) \Rightarrow (v_1 = v_2)$ ensures that v_0 has at most one outgoing f -edge. The neighborhood formula $(v_0 \xrightarrow{f} v_1) \wedge (v_0 \xrightarrow{f} v_2)$ contains two edges emanating from the central node v_0 . The restriction on the neighborhood is that the edges are in fact the same, because they have the same source, v_0 , the same target, $v_1 = v_2$, and the same label f .

Shorthands

We use $c[R]p$ to denote a reachability constraint (4.1). Intuitively, the reachability constraint requires that every node that is reachable from c by following an R -path satisfy the pattern p .

We use **let** expressions to specify the scope in which the pattern is declared:

$$\mathbf{let} \ p_1(v_0) \stackrel{\text{def}}{=} N_1(v_0, \dots, v_n) \Rightarrow \psi_1(v_0, \dots, v_n) \ \mathbf{in} \ \varphi$$

This allows us to write more concise formulas via reuse of pattern definitions. For example, we can say that program variables x and y are pointing to (potentially shared) doubly-linked lists:

$$\mathbf{let} \ inv_{f,b}(v_0) \stackrel{\text{def}}{=} (v_0 \xrightarrow{f} v_1 \Rightarrow v_1 \xrightarrow{b} v_0) \ \mathbf{in} \ x[\xrightarrow{f}^*]inv_{f,b} \wedge y[\xrightarrow{f}^*]inv_{f,b}$$

We use $c_1[R]\neg c_2$ to denote $\mathbf{let} \ p(v_0) \stackrel{\text{def}}{=} (true \Rightarrow \neg(v_0 = c_2)) \ \mathbf{in} \ c_1[R]p$. In this simple case, the neighborhood is only the node assigned to v_0 . Intuitively, $c_1[R]\neg c_2$ means that the node labeled by constant c_2 is not reachable along an R -path from the node labeled by c_1 . We use $c_1 \langle R \rangle c_2$ as a shorthand for $\neg(c_1[R]\neg c_2)$. Intuitively, $c_1 \langle R \rangle c_2$ means that *there exists* an R -path from c_1 to c_2 . We use $c_1 = c_2$ to denote $c_1 \langle \epsilon \rangle c_2$, and $c_1 \neq c_2$ to denote $\neg(c_1 = c_2)$.

We use $c[R](p_1 \wedge p_2)$ to denote $(c[R]p_1) \wedge (c[R]p_2)$, when p_1 and p_2 agree on the central node variable. When two patterns are often used together, we introduce a name for their conjunction (instead of naming each one separately): $\mathbf{let} \ p(v_0) \stackrel{\text{def}}{=} (N_1 \Rightarrow \psi_1) \wedge (N_2 \Rightarrow \psi_2) \ \mathbf{in} \ \varphi$.

For a quantifier-free formula $\psi(v_0)$ with a single free variable v_0 , we write $c[R]\psi$ instead of $\mathbf{let} \ p(v_0) \stackrel{\text{def}}{=} (true \Rightarrow \psi(v_0)) \ \mathbf{in} \ c[R]p$. In particular, for a unary relation symbol u , we use $c[R]u$ to denote $\mathbf{let} \ p(v_0) \stackrel{\text{def}}{=} (true \Rightarrow u(v_0)) \ \mathbf{in} \ c[R]p$. We use $u(c)$ to denote the formula $c \langle \epsilon \rangle u$ (equivalently, $c \langle \epsilon \rangle u$). We abuse the notations slightly by writing $N \wedge \psi_1 \Rightarrow \psi_2$ instead of $N \Rightarrow (\psi_1 \Rightarrow \psi_2)$.

In routing expressions, we use $\underline{\Sigma}$ to denote the routing expression $(\underline{f}_1 | \underline{f}_2 | \dots | \underline{f}_m)$, the union of all the fields in F . Similarly, $\overline{\Sigma}$ denotes the routing expression $(\overline{f}_1 | \overline{f}_2 | \dots | \overline{f}_m)$. For example, $c_1[\underline{\Sigma}^*]\neg c_2$ means that c_2 is not reachable from c_1 by any path. Finally, we sometimes omit the concatenation operator “.” in routing expressions.

4.1.2 Semantics of \mathcal{L}_0

\mathcal{L}_0 formulas are interpreted over labeled directed graphs. A labeled directed graph G over a vocabulary $\tau = \langle C, U, F \rangle$ is a tuple $\langle V^G, E^G, C^G, U^G \rangle$ where:

- V^G is a set of nodes modelling the heap objects,
- $E^G: F \rightarrow \mathcal{P}(V^G \times V^G)$ are labeled edges,
- $C^G: C \rightarrow V^G$ provides interpretation of constants as unique labels on the nodes of the graph, and

² In all our examples, a neighborhood formula N used in a pattern is such that B_N (the Gaifman graph of N) is connected.

- $U^G: U \rightarrow \mathcal{P}(V^G)$ maps unary relation symbols to the set of nodes in which they hold.

The language $L(R)$ of words accepted by a routing expression R is defined as usual for regular expression. The semantics of \mathcal{L}_0 formulas is formally defined as follows.

Definition 4.1.5 Consider a routing expression R and $w \in L(R)$. We say that **there is a path labeled by w from a node s_1 to a node s_2 in G** if one of the following conditions holds:

- $s_1 = s_2$ and $w = \epsilon$,
- $s_1 = s_2$, $w = u$ for a unary relation symbol u and $s_1 \in U^G(u)$,
- $s_1 = s_2$, $w = \neg u$ for a unary relation symbol u and $s_1 \notin U^G(u)$,
- $s_1 = s_2$, $w = c$ for a constant c and $C^G(c) = s_1$,
- $s_1 = s_2$, $w = \neg c$ for a constant c and $C^G(c) \neq s_1$,
- $w = \underline{f}$ for an edge $f \in F$ and $\langle s_1, s_2 \rangle \in E^G(f)$,
- $w = \overline{f}$ for an edge $f \in F$ and $\langle s_2, s_1 \rangle \in E^G(f)$,
- $w = w_1.w_2$ and there exists a node s_3 such that there is a path labeled by w_1 from s_1 to s_3 and there exists a path labeled by w_2 from s_3 to s_2 .

A node tuple in G satisfies a pattern p if it satisfies the quantifier-free formula that defines p , according to the usual semantics of the first-order logic over graph structures.

The satisfaction relation \models between a graph G and an \mathcal{L}_0 -formula is defined similarly to the usual semantics of the first-order logic with transitive closure over graphs. A graph G satisfies a formula $c[R]p$ (and we write $G \models c[R]p$) if and only if for every $w \in L(R)$ and for every node tuple s_0, \dots, s_n in G , if there is a path labeled by w from c to s_0 , then the tuple s_0, \dots, s_n , satisfies p with s_0 used as the central node for p . The meaning of Boolean connectives is defined in a standard manner.

We say that node $s \in G$ is labeled with σ if $\sigma \in C$ and $s = C^G(\sigma)$ or $\sigma \in U$ and $s \in U^G(\sigma)$. For an edge $\langle s_1, s_2 \rangle \in G$ and $f \in F$, we say that the edge $\langle s_1, s_2 \rangle$ is labeled with f , if $\langle s_1, s_2 \rangle \in E^G(f)$. In the rest of this chapter, *graph* denotes a directed labeled graph, in which nodes are labeled by constant and unary relation symbols, and edges are labeled by binary relation symbols, as defined above.

Remark. The translation from \mathcal{L}_0 to MSO in Section 4.4.1 provides an alternative definition for the semantics of \mathcal{L}_0 .

4.1.3 Finite Model Property

We are interested in checking validity (and satisfiability) of \mathcal{L}_0 formulas only over finite graphs. The graphs are finite because they represent data-structures allocated by a program. (However, the graphs may be unbounded, due to dynamic allocation of memory.) In general, a finite validity problem is considered more difficult than a validity problem. For example, in first-order logic, the validity problem is recursively enumerable while the finite validity problem is not. In a logic with the finite model property, the notions of validity and *finite* validity coincide. Thus, the finite model property is desirable.

\mathcal{L}_0 with arbitrary patterns has a finite model property. If formula $\varphi \in \mathcal{L}_0$ has an infinite model, each reachability constraint in φ that is satisfied by this model has a finite witness.

Theorem 4.1.6 (Finite model property) Every satisfiable \mathcal{L}_0 formula is satisfiable by a finite graph.

Sketch of Proof: We show that \mathcal{L}_0 can be translated into a fragment of an infinitary logic that has a finite model property. Observe that $c[R]p$ is equivalent to an infinite conjunction of universal first-order sentences. Therefore, if G is a model of $c[R]p$ then every subgraph of G is also its model. Dually, $\neg c[R]p$ is equivalent to an infinite disjunction of existential first-order sentences. Therefore, if G is a model of $\neg c[R]p$, then G has a finite subgraph G' such that every subgraph of G that contains G' is a model of $\neg c[R]p$. It follows that every satisfiable boolean combination of formulas of the form $c[R]p$ has a finite model. Thus, \mathcal{L}_0 has a finite model property.

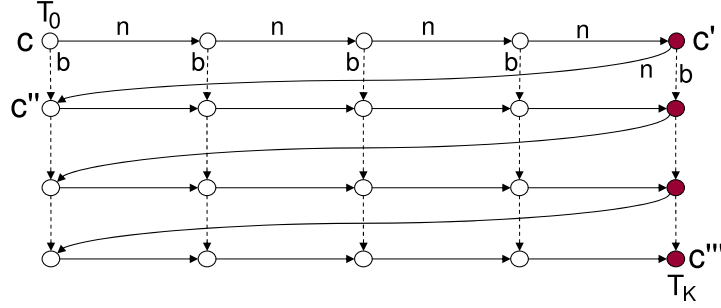


Figure 4.1: A sketch of a grid model for a tiling problem \mathcal{T} . The n -edges are depicted with solid lines, the b -edges are depicted with dashed lines. The filled circles denote nodes labeled with “red”.

4.2 Undecidability of \mathcal{L}_0

The satisfiability and the validity problems of \mathcal{L}_0 formulas are undecidable. Since \mathcal{L}_0 is closed under negation, it is sufficient to show that its satisfiability problem is undecidable. The proof uses a reduction from the tiling problem.

Definition 4.2.1 Define a *tiling problem*, $\mathcal{T} = \langle T, R, D \rangle$, to consist of a finite list of tile types, $T = [t_0, \dots, t_k]$, together with horizontal and vertical adjacency relations, $R, D \subseteq T^2$. Here $R(a, b)$ means that tiles of type b fit immediately to the right of tiles of type a , and $D(a, b)$ means that tiles of type b fit one step down from those of type a . A solution to a tiling problem is an arrangement of instances of the tiles in a rectangular grid such that a t_0 tile occurs in the top left node of the grid, and a t_k tile occurs in the bottom right node of the grid, and all adjacency relationships are respected.

It is well-known that tiling problems of this flavor are undecidable. Therefore, if a logic can express tilings, its satisfiability problem is also undecidable. Given a tiling problem \mathcal{T} , we construct a formula $\varphi_{\mathcal{T}}$, such that $\varphi_{\mathcal{T}}$ is satisfiable if and only if there exists a solution to \mathcal{T} .

The idea is that each node in the graph that satisfies $\varphi_{\mathcal{T}}$ describes a tile, with unary relation symbols T_0, \dots, T_k encoding the tile types t_0, \dots, t_k . There is a b -edge between every two nodes that are vertically adjacent in the grid. There is an n -edge between every two nodes that are horizontally adjacent in the grid, and from the last node of every row to the first node in the subsequent row. The constant c labels the top left node of the grid, the constant c' labels the top right node of the grid, the constant c'' labels the first node of the second row of the grid, and the constant c''' labels the bottom right node of the grid (see sketch in Fig. 4.1). The unary relation *red* labels the nodes of the last column of the grid.

The most interesting part of the formula $\varphi_{\mathcal{T}}$ ensures that all graphs that satisfy $\varphi_{\mathcal{T}}$ have a grid-like form. It states that for every node v that is n -reachable from c , if there is a b -edge from v to u , then there is a b -edge from the n -successor of v to the n -successor of u :

$$\mathbf{let} \quad p(v) \stackrel{\text{def}}{=} (v \xrightarrow{b} u) \wedge (v \xrightarrow{n} v_1) \wedge (u \xrightarrow{n} u_1) \Rightarrow (v_1 \xrightarrow{b} u_1) \quad \mathbf{in} \quad c[(\xrightarrow{n})^*]p \quad (4.2)$$

Theorem 4.2.2 (Undecidability) *The satisfiability problem of \mathcal{L}_0 formulas is undecidable.*

Proof: Given a tiling problem $\mathcal{T} = \langle T, R, D \rangle$, we construct an \mathcal{L}_0 formula $\varphi_{\mathcal{T}}$ as a conjunction of the following formulas:

1. There is n -path from c to c' : $c[(\xrightarrow{n})^*]c'$
2. There is n -edge from c' to c'' : $c'[\xrightarrow{n}]c''$
3. There is n -path from c'' to c''' : $c''[(\xrightarrow{n})^*]c'''$
4. There is b -edge from c to c'' : $c[\xrightarrow{b}]c''$.
5. No n -edge exits t : $c'''[\xrightarrow{n}]false$.

6. For every node v that is n -reachable from s , if there is a b -edge from v to u , then there is a b -edge from the n -successor of v to the n -successor of u : $\mathbf{let} p(v) \stackrel{\text{def}}{=} (v \xrightarrow{b} u) \wedge (v \xrightarrow{n} v_1) \wedge (u \xrightarrow{n} u_1) \Rightarrow (v_1 \xrightarrow{b} u_1) \mathbf{in} c[(\xrightarrow{n})^*]p$.
7. The n -edges and the b -edges reachable from s are deterministic: $\mathbf{let} det_n(v) \stackrel{\text{def}}{=} (v \xrightarrow{n} v') \wedge (v \xrightarrow{n} v'') \Rightarrow (v' = v'') \mathbf{in} s[(\xrightarrow{n})^*]det_n$, similarly, for b -edges.
8. The top left node of the grid has a t_0 tile type, and the bottom right node of the grid has a t_k tile type: $T_0(c) \wedge T_k(c''')$.
9. Each node in the grid has exactly one tile type:

$$c[(\xrightarrow{n})^*] \left(\bigwedge_{0 \leq i < j \leq k} \neg(T_i \wedge T_j) \right) \wedge \left(\bigvee_{0 \leq i \leq k} T_i \right)$$

10. Every node in the last column of the grid is labeled with red : $c'[(\xrightarrow{b})^*]red$.
11. To express that only nodes in the last column of the grid are labeled with red , we say that the first row is not labeled with red , except its last node, and if a node is labeled with red , then its b -predecessor is labeled:

$$c[(\xrightarrow{n}, \neg c')^*] \neg red \wedge \mathbf{let} p(v) \stackrel{\text{def}}{=} (w \xrightarrow{b} v) \wedge red(v) \Rightarrow red(w) \mathbf{in} c[(\xrightarrow{n})^*]p$$

12. Two horizontally adjacent tiles are compatible according to R :

$$\mathbf{let} p(v) \stackrel{\text{def}}{=} (v \xrightarrow{n} w) \wedge \neg red(v) \Rightarrow \left(\bigvee_{R(t_i, t_j)} (T_i(v) \wedge T_j(w)) \right) \mathbf{in} c[(\xrightarrow{n})^*]p$$

13. Two vertically adjacent tiles are compatible according to D :

$$\mathbf{let} p(v) \stackrel{\text{def}}{=} (v \xrightarrow{b} w) \Rightarrow \bigvee_{D(t_i, t_j)} (T_i(v) \wedge T_j(w)) \mathbf{in} c[(\xrightarrow{n})^*]p$$

Remark. The reduction uses only two binary relation symbols and a fixed number of unary relation symbols. It can be modified to show that the logic with three binary relation symbols (and no unary relations) is undecidable.

4.3 Decidable and Useful Fragment of \mathcal{L}_0

In this section, we define a fragment of \mathcal{L}_0 , called \mathcal{L}_1 , by syntactically restricting the patterns. We show that \mathcal{L}_1 naturally describes some commonly-used data-structures, express verification conditions, and characterizes certain shape abstractions. In the next section, we show that \mathcal{L}_1 is decidable.

4.3.1 The \mathcal{L}_1 Fragment

The \mathcal{L}_1 fragment is defined by syntactically restricting the patterns which can be used. The fragment \mathcal{L}_1 permits arbitrary boolean combinations in patterns, but it restricts the distance between variables and forbids the use of constants in positive occurrences of equality and edge formulas.

Definition 4.3.1 (The syntax of \mathcal{L}_1) *In every reachability constraint $c[R]p$ that appears in an \mathcal{L}_1 formula, the pattern $p(v_0) \stackrel{\text{def}}{=} N(v_0, \dots, v_n) \Rightarrow \psi(v_0, \dots, v_n)$ satisfies the following restrictions on ψ :*

- **(equality restriction)** *If ψ contains a positive occurrence of an equality between variables $v_i = v_j$, then the distance between v_i and v_j in N is at most 2 (distance is defined in Definition 4.1.2).*

Pattern Name	Pattern Definition	Meaning
$det_f(v_0)$	$(v_0 \xrightarrow{f} v_1) \wedge (v_0 \xrightarrow{f} v_2) \Rightarrow (v_1 = v_2)$	at most one outgoing f -edge from v_0
$uns_f(v_0)$	$(v_1 \xrightarrow{f} v_0) \wedge (v_2 \xrightarrow{f} v_0) \Rightarrow (v_1 = v_2)$	v_0 has at most one incoming f -edge
$uns_{f,g}(v_0)$	$(v_1 \xrightarrow{f} v_0) \wedge (v_2 \xrightarrow{g} v_0) \Rightarrow false$	v_0 is not heap-shared by f -edge and g -edge
$inv_{f,b}(v_0)$	$(v_0 \xrightarrow{f} v_1 \Rightarrow v_1 \xrightarrow{b} v_0)$	every f -edge from v_0 to v_1 has a b -edge in the opposite direction.
$same_{f,g}(v_0)$	$(v_0 \xrightarrow{f} v_1 \Rightarrow v_0 \xrightarrow{g} v_1) \wedge (v_0 \xrightarrow{g} v_1 \Rightarrow v_0 \xrightarrow{f} v_1)$	edges f and g emanating from v_0 are parallel

Figure 4.2: Useful pattern definitions ($f, b, g \in F$ are edge labels).

- (**edge restriction**) If ψ contains a positive occurrence of an edge formula of the form $v_i \xrightarrow{f} v_j$, then the distance between v_i and v_j in N is at most 1.
- (**constant restriction**) Positive occurrences of formulas of the form $v \xrightarrow{f} c$, $c \xrightarrow{f} v$, and $v = c$ in ψ are not allowed.

Remark. Note that formula (4.2), which is used in the proof of undecidability in Theorem 4.2.2, is not in \mathcal{L}_1 , because p contains a positive $v_1 \xrightarrow{b} u_1$ with distance 3 between v_1 and u_1 , while \mathcal{L}_1 allows edge patterns with distance at most 1.

4.3.2 Describing Linked Data-Structures in \mathcal{L}_1

In this section, we show that \mathcal{L}_1 can express properties of data-structures. Fig. 4.2 lists some useful patterns and their meanings. For example, the first pattern det_f means that there is at most one outgoing f -edge from a node. Another important pattern uns_f means that a node has at most one incoming f -edge. We use the subscript f to emphasize that this definition is parametric in f .

Well-formed heaps We assume that C (the set of constant symbols) contains a constant for each pointer variable in the program (denoted by x, y in our examples). Also, C contains a designated constant $null$ that represents null values. Throughout the rest of the chapter we assume that all the graphs denote well-formed heaps, i.e., the fields of all objects reachable from constants are deterministic, and dereferencing NULL yields $null$. In \mathcal{L}_1 this is expressed by the formula:

$$WF \stackrel{\text{def}}{=} \left(\bigwedge_{c \in C} \bigwedge_{f \in F} c[\Sigma^*] det_f \right) \wedge \left(\bigwedge_{f \in F} null \langle \xrightarrow{f} \rangle null \right) \quad (4.3)$$

Using the patterns in Fig. 4.2, Fig. 4.3 defines some interesting properties of data-structures using \mathcal{L}_1 . The formula $reach_{x,f,y}$ means that the object pointed-to by the program variable y is reachable from the object pointed-to by the program variable x by following an access path of f field pointers. We can also use it with $null$ in the place of y . For example, the formula $reach_{x,f,null}$ describes a (possibly empty) linked-list pointed-to by x . Note that $reach_{x,f,null}$ implies that the list is acyclic, because $null$ is always a “sink” node in a well-formed heap. We can also express that there are no incoming f -edges into the list pointed to by x , by conjoining the previous formula with $unshared_{x,f}$. We can specify the fact that x is located on a cycle of f -edges: $cyclic_{x,f}$. Disjointness can be expressed by the formula $disjoint_{x,f,y,g}$ that uses both forward and backward traversal of edges in the routing expression. Disjointness of data-structures is important for parallelization (e.g., see [HHN92]). For example, we can express that the linked list pointed to by x is disjoint from the linked-list pointed to by y , using the formula $disjoint_{x,f,y,f}$. This formula guarantees that every node v that is reachable from the node pointed-to by x using an f -path must *not* be reachable from y using an f -path. However, v may be reachable from y using other edges, or v may be a part of another data-structure which shares elements with y .

The last three examples in Fig. 4.3 specify data-structures with multiple fields. The formula $inverse_{x,f,b,y}$ describes a doubly-linked list with variables x and y pointing to the head and the tail of the list, respectively. First, it guarantees the existence of an f -path. Next, it uses the pattern $inv_{f,b}$ to express that if there is an f -edge from one node to another, then there is a b -edge in the opposite direction. This pattern is applied to all nodes on the f -path that starts from x and that does not visit y , expressed using the test “ $\neg y$ ” in the routing expression.

Name	Formula
$reach_{x,f,y}$	$x \langle (\underline{f})^* \rangle y$ the heap object pointed-to by y is reachable from the heap object pointed-to by x .
$cyclic_{x,f}$	$x \langle (\underline{f})^+ \rangle x$ cyclicity: the heap object pointed-to by x is located on a cycle.
$unshared_{x,f}$	$x [(\underline{f})^*] uns_f$ every heap object reachable from x by an f -path has at most one incoming f -edge.
$disjoint_{x,f,y,g}$	$\neg(x \langle (\underline{f})^* \rangle (\underline{g})^* y)$ disjointness: there is no heap object that is reachable from x by an f -path and also reachable from y by a g -path.
$same_{x,f,g}$	$x [(\underline{f}) \mid \underline{g}]^* same_{f,g}$ the f -path and the g -path from x are parallel, and traverse the same objects.
$inverse_{x,f,b,y}$	$reach_{x,f,y} \wedge x [(\underline{f}, \neg y)^*] inv_{f,b}$ doubly-linked lists between two variables x and y with f and b as forward and backward edges.
$tree_{root,r,l}$	$root [(\underline{l} \mid \underline{r})^*] (uns_{l,r} \wedge uns_l \wedge uns_r) \wedge \neg (root \langle (\underline{l} \mid \underline{r})^+ \rangle root)$ tree rooted at $root$.
$tree_{root,r,l,b}$	$tree_{root,r,l} \wedge root [(\underline{l} \mid \underline{r})^*] inv_{l,b} \wedge inv_{r,b}$ tree rooted at $root$ with parent pointers b from every tree node to its parent.

Figure 4.3: Properties of data-structures expressed in \mathcal{L}_1 .

The formula $tree_{root,r,l}$ describes a binary tree. The first part requires that the nodes reachable from the root (by following any path of l and r fields) not be heap-shared. The second part prevents edges from pointing back to the root of the tree by forbidding the root to participate in a cycle. The formula $tree_{root,r,l,b}$ describes a binary tree rooted at $root$ with parent pointers b from every tree node to its parent.

The ability to express properties like $tree_{root,r,l}$ is non-trivial, because we are operating on general graphs, and not just trees. Operating on general graphs allows us to verify that the data-structure invariant is reestablished after a sequence of low-level mutations that temporarily violate the data-structure invariant.

Unary relations symbols can be used to describe data values from a limited domain, and their interaction with the structural properties of the heap. For example, for a tree we can specify that both children of every white node are green:

```

let  $wg(v_0) \stackrel{\text{def}}{=} (white(v_0) \wedge (v_0 \underline{l} v_1) \Rightarrow green(v_1)) \wedge (white(v_0) \wedge (v_0 \underline{r} v_1) \Rightarrow green(v_1))$ 
in  $root [(\underline{l} \mid \underline{r})^*] wg$ 

```

Moreover, unary information can be used to describe states of objects, and sets of objects.

4.3.3 Expressing Verification Conditions in \mathcal{L}_1

The Reverse Procedure

The reverse procedure shown in Fig. 4.4 performs in-place reversal of a singly-linked list. This procedure is interesting because it destructively updates the list and a natural specification of its partial correctness requires reasoning about two fields. Moreover, it manipulates linked lists in which each list node can be pointed-to from the outside. We show that the verification conditions for the procedure `reverse` can be expressed in \mathcal{L}_1 . If the verification conditions are valid, then the program is partially correct with respect to the specification. The validity of the verification conditions can be checked automatically because the logic \mathcal{L}_1 is decidable, as shown in the next section. We can show how to automatically generate verification conditions in \mathcal{L}_1 for arbitrary procedures that are annotated with preconditions, postconditions, and loop invariants in \mathcal{L}_1 .

```

Node reverse(Node x){
  [0] Node y = null;
  [1] while (x != null){
  [2]   Node t = x.n;
  [3]   x.n = y;
  [4]   y = x;
  [5]   x = t;
  [6] }
  [7] return y;
}

```

Figure 4.4: The reverse procedure performs in-place reversal of a singly-linked list

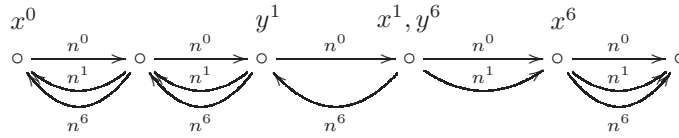


Figure 4.5: An example graph that satisfies the VC_{loop} formula for `reverse`.

Notice that in this section we assume that all graphs denote valid states, i.e., satisfy (4.3). The precondition requires that x point to an acyclic list, on entry to the procedure. We use the symbols x^0 and n^0 to record the values of the variable x and the n -field on entry to the procedure.

$$pre_{reverse} \stackrel{\text{def}}{=} x^0 \langle (n^0)^* \rangle null$$

The postcondition ensures that the result is an acyclic list pointed-to by y . Most importantly, it ensures that each edge of the original list is reversed in the returned list, which is expressed in a similar way to a doubly-linked list, using *inverse* formula. We use the relation symbols y^7 and n^7 to refer to the values on exit.

$$post_{reverse} \stackrel{\text{def}}{=} y^7 \langle (n^7)^* \rangle null \wedge inverse_{x^0, n^0, n^7, y^7}$$

The loop invariant φ shown below relates the heap on entry to the procedure to the heap at the beginning of each loop iteration (line [1]). First, we require that the part of the list reachable from x be the same as it was on entry to `reverse`. Second, the list reachable from y is reversed from its initial state. Finally, the only original edge outgoing of y is to x .

$$\varphi \stackrel{\text{def}}{=} same_{x^1, n^0, n^1} \wedge inverse_{x^0, n^0, n^1, y^1} \wedge y^1 \langle (n^0)^* \rangle x^1$$

Note that the postcondition uses two binary relations, n^0 and n^7 , and also the loop invariant uses two binary relations, n^0 and n^1 . This illustrates that reasoning about singly-linked lists may involve more than one binary relation.

The verification condition of `reverse` consists of two parts, VC_{loop} and VC_{exit} , explained below.

The formula VC_{loop} expresses the fact that φ is indeed a loop invariant. To express it in our logic, we use several copies of the vocabulary, one for each program point. Different copies of the relation symbol n in the graph model values of the field n at different program points. Similarly, for constants. For example, Fig. 4.5 shows a graph that satisfies the formula VC_{loop} below. It models a heap at the end of some loop iteration of `reverse`. The superscripts of the symbol names denote the corresponding program points.

To show that the loop invariant φ is maintained after executing the loop body, we assume that the loop condition and the loop invariant hold at the beginning of the iteration, and show that the loop body was executed

```

Node append(Node x, Node y) {
  [0] Node t = x;
  [1] if (t == null)
  [2]   return y;
  [3] while (t.n != null) {
  [4]   t = t.n;
  [5] }
  [6] t.n = y;
  [7] return x;
}

```

Figure 4.6: The append procedure concatenates two singly-linked lists.

without performing a null-dereference, and the loop invariant holds at the end of the loop body:

$$\begin{array}{ll}
VC_{loop} \stackrel{\text{def}}{=} (x^1 \neq null) & \text{loop is entered} \\
\wedge \varphi & \text{loop invariant holds on loop head} \\
\wedge (y^6 = x^1) \wedge x^1 \langle n^1 \rangle x^6 \wedge x^1 \langle n^6 \rangle y^1 & \text{loop body} \\
\wedge \text{same}_{y^1, n^1, n^6} \wedge \text{same}_{x^6, n^1, n^6} & \text{rest of the heap remains unchanged} \\
\Rightarrow (x^1 \neq null) & \text{no null-dereference in the body} \\
\wedge \varphi^6 & \text{loop invariant after executing loop body}
\end{array}$$

Here, φ^6 denotes the loop-invariant formula φ after executing the loop body (line [6]), i.e., replacing all occurrences of x^1 , y^1 and n^1 in φ by x^6 , y^6 and n^6 , respectively. The formula VC_{loop} defines a relation between three states: on entry to the procedure, at the beginning of a loop iteration and at the end of a loop iteration.

The formula VC_{exit} expresses the fact that if the precondition holds and the execution reaches the exit of the procedure (i.e., the loop is not entered because the loop condition does not hold), the postcondition holds on exit: $VC_{exit} \stackrel{\text{def}}{=} pre \wedge (x^0 = x^1) \wedge (x^1 = null) \Rightarrow post$.

The Append Procedure

The append procedure given in Fig. 4.6 concatenates two singly-linked lists.

To describe the effect of a procedure on the heap, we sometimes use *auxiliary* relations and constants, whose interpretation is constrained in the precondition, and used in the postconditions. It allows us to relate the values after a call to a procedure returns to the values before that call. Note that the auxiliary constant does not have an index, because it is not part of the program. In this example, we use the auxiliary constant *last* to label the last node of the first list.

The precondition for append requires that x and y point to acyclic and disjoint lists, and defines the meaning of the new constant *last*:

$$\begin{aligned}
pre_{append} = & x^0 \langle (n^0)^* \rangle null \wedge y^0 \langle (n^0)^* \rangle null \wedge disjoint_{x^0, n^0, y^0, n^0} \wedge \\
& x^0 \langle (n^0, \neg null)^* \rangle last \wedge last \langle n^0 \rangle null
\end{aligned}$$

The postcondition for append uses x^7 to denote the return value, which points to an acyclic list. It uses the constant *last* to identify the object whose *next* field was modified by the procedure.

$$\begin{aligned}
post_{append} = & x^7 \langle (n^7)^* \rangle null \wedge x^7 = x^0 \wedge last \langle n^7 \rangle y^0 \wedge \\
& x^0 \langle (n^0, \neg last)^* \rangle same_{n^0, n^7} \wedge y^0 \langle (n^0)^* \rangle same_{n^0, n^7}
\end{aligned}$$

4.3.4 Characterizing Shape Abstractions in \mathcal{L}_1

Recall that $\hat{\gamma}$ operation maps every abstract value a of a given abstract domain to a logical formula, called a *characteristic formula*, whose meaning is exactly the set $\gamma(a)$. Specifically, [Yor03, YRSW07], gives an algorithm for $\hat{\gamma}$ that characterizes canonical abstraction [SRW02] using first-order logic with transitive closure. The problem

is that automatic reasoning in first-order logic with transitive closure is difficult, as discussed in Section 1.3.4. Instead, we can use a decidable logic \mathcal{L}_1 to characterize certain shape abstractions such as [MYRS05, LAIS06]. In particular, we show in this section how to characterize the shape abstraction described in [MYRS05].

The abstraction of [MYRS05] is designed for programs operating on singly-linked lists. The idea is to summarize list elements on unshared list segments not pointed-to by local variables. An object is an *interruption* if it is pointed-to by a variable (or null) or heap-shared (i.e., has two or more predecessors). An *uninterrupted list* is a path delimited by two interruptions that does not contain interruptions other than the delimiters.

The abstraction of a concrete state is performed in three steps:

- (a) remove all garbage objects (i.e., objects not reachable from any program variable),
- (b) partition the heap into uninterrupted lists, where each list is delimited by a pair of interruption objects,
- (c) abstract the path length of the uninterrupted lists into “=1” (exactly one edge) and “>1” (more than one edge).

It is easy to describe an uninterrupted list in \mathcal{L}_1 . The main difficulty in specifying $\hat{\gamma}$ in \mathcal{L}_1 is that not all interruption objects have unique names. That is, an interruption object can be a heap-shared object that is not pointed-to by a program variable. Fortunately, the restriction to singly-linked lists allows us to uniquely identify each of these objects by its distance (defined by the number of uninterrupted lists) from objects pointed-to by program variables. For every program variable x , we count the interruption objects on the (unique) path emanating from x , and mark the i -th interruption with an auxiliary variable x_i . We need at most n^2 auxiliary variables to mark all interruptions, because every simple path in a garbage-free heap, consisting of only singly-linked lists with n program variables, contains at most n interruptions.

Logical representation of concrete states Given a set of program variables Var , where $n = |Var|$, we define the vocabulary $\tau = \{C, U, F\}$ where $C = \{x_i \mid x \in Var, 0 \leq i \leq n\} \cup Var \cup \{null\}$, U is empty, and $F = \{f\}$. A graph G over τ represents a valid concrete state when G represents a well-formed heap (i.e., satisfies the formula WF given in (4.3)), and for every $x \in Var$, and for every $i = 0, \dots, n$, the constant symbol x_i is interpreted as the i -th interruption on the unique path from the object pointed-to by a variable x . In particular, x and x_0 are interpreted by the same node in G . If there are less than n interruptions reachable from x , then the remaining auxiliary variables are interpreted by the same node as *null*.

Characterization of abstraction An abstract value is a set of shape graphs. A shape graph S , as defined in [MBC+07], is a quadruple $\langle Nodes^S, Edges^S, Env^S, Len^S \rangle$, where $Nodes$ is the set of nodes, which represent the interruption objects and the designated *null* node, $Edges^S \subseteq Nodes^S \times Nodes^S$ is a set of edges, each of which represents an uninterrupted list, $Env^S: Var \cup \{null\} \rightarrow Nodes^S$ maps program variables (and null) to nodes, and $Len^S: Edges^S \rightarrow \{=1, >1\}$ maps edges to their (abstracted) lengths. We omit the superscript S when no confusion is likely.

The $\hat{\gamma}$ operation returns an \mathcal{L}_1 formula over τ , computed as described below.

For every shape graph $\langle Nodes, Edges, Env, Len \rangle$, we compute the mapping $m: C \rightarrow Nodes$ as follows:

$$\begin{aligned}
 m(null) &:= Env(null) \\
 \text{For every } x \in Var & \\
 m(x) &:= Env(x) \\
 m(x_0) &:= Env(x) \\
 \text{For every } x \in Var, \text{ For every } i = 1, \dots, n-1 & \\
 \text{If } m(x_i) = m(null) \text{ then } m(x_{i+1}) &:= m(null) \\
 \text{else } m(x_{i+1}) &:= v' \text{ where } (m(x_i), v') \in Edges
 \end{aligned}$$

The mapping m extends Env with auxiliary variables. For $i = 1, \dots, n$, we define the formula $\xi[x_i]$ which characterizes uninterrupted lists between auxiliary variables x_{i-1} and x_i , where the length of a list is given by Len of the corresponding edge:

$$\xi[x_i] \stackrel{\text{def}}{=} \begin{cases} x_{i-1} \langle \underline{f} \rangle x_i & \text{if } Len(m(x_{i-1}), m(x_i)) \text{ is } =1 \\ x_{i-1} \langle \underline{f}, \underline{f}^+ \rangle x_i \wedge x_{i-1} [(\underline{f}, \neg x_i)^+] \text{uns}_f \wedge \text{npt} & \text{if } Len(m(x_{i-1}), m(x_i)) \text{ is } >1 \end{cases}$$

To enforce that the intermediate nodes on the path from x_{i-1} to x_i are unshared, we use the pattern uns_f , defined as in Fig. 4.2: $\text{uns}_f(v_0) \stackrel{\text{def}}{=} (v_1 \underline{f} v_0) \wedge (v_2 \underline{f} v_0) \Rightarrow (v_1 = v_2)$. To enforce that the intermediate nodes are not

pointed-to by any program or auxiliary variable, we use the pattern npt defined by:

$$npt(v_0) \stackrel{\text{def}}{=} \bigwedge_{c \in C} c \neq v_0$$

For every pair of variables $c_1, c_2 \in C$, we use $\xi_{eq}[c_1, c_2]$ to denote the formula $c_1 = c_2$ if $m(c_1) = m(c_2)$, and the formula $c_1 \neq c_2$ otherwise.

Now we can define a formula that characterizes a shape graph:

$$\xi(S) \stackrel{\text{def}}{=} WF \wedge \bigwedge_{\substack{c_1, c_2 \in C \\ c_1 \neq c_2}} \xi_{eq}[c_1, c_2] \wedge \bigwedge_{\substack{x \in \text{Var} \\ i = 0, \dots, n}} \xi[x_i]$$

Note that in every graph G that satisfies $\xi(S)$, every x_i is interpreted as the i -th interruption from an object pointed-to by a variable x . That is, two graphs that satisfy $\xi(S)$ cannot differ only by their interpretation of the auxiliary variables.

For every abstract value a , $\hat{\gamma}(a)$ is a disjunction of characteristic formulas for the shape graphs that constitute a : $\hat{\gamma}(a) \stackrel{\text{def}}{=} \bigvee_{S \in a} \xi(S)$.

The $\hat{\gamma}$ operation defined above exactly characterizes the abstraction of [MYRS05]. That is, for every abstract value a , and every graph G over τ , $G \models \hat{\gamma}(a)$ if and only if the abstraction of the concrete state represented by G is a shape graph in a .

To simplify the exposition, we use a straightforward encoding of the $\hat{\gamma}$. It is possible to optimize this encoding, in particular, by taking into account the context in which the $\hat{\gamma}$ formulas are used.

4.4 Decidability of \mathcal{L}_1

In this section, we show that \mathcal{L}_1 is decidable for validity and satisfiability. Since \mathcal{L}_1 is closed under negation, it is sufficient to show that it is decidable for satisfiability. The proof proceeds as follows:

1. Translate an \mathcal{L}_0 formula into an equivalent formula in weak monadic second-order (MSO) logic (Lemma 4.4.2).
2. Define a class of simple graphs \mathcal{A}_k , for which the Gaifman graph (Definition 4.4.4) is a tree with at most k additional edges (Definition 4.4.5).
3. Show that the satisfiability of MSO logic over \mathcal{A}_k is decidable, by reduction to MSO logic on trees [Rab69] (Lemma 4.4.6). We could have also shown decidability using the fact that the tree width of all graphs in \mathcal{A}_k is bounded by k , and that MSO logic over graphs with bounded tree width is decidable [Cou89, ALS91, See92].
4. Every formula $\varphi \in \mathcal{L}_1$ can be effectively translated into an equi-satisfiable normal-form formula that is a disjunction of formulas in $C\mathcal{L}_1$ (Definition 4.4.9 and Theorem 4.4.12). It is sufficient to show that the satisfiability of $C\mathcal{L}_1$ is decidable.
5. Show that if formula $\varphi \in C\mathcal{L}_1$ has a model, φ has a model in \mathcal{A}_k , where k is proportional to the size of the formula φ (Theorem 4.4.14). This is the main part of the proof, given in detail in Section 4.5.

In Section 4.6, we extend this proof to show decidability of \mathcal{L}_2 .

4.4.1 Translation from \mathcal{L}_0 to MSO

Every regular expression R can be effectively translated into an MSO formula $\varphi_R(x, y)$, that describes the paths from x to y labeled with w , for every word w in R . To encode the Kleene star expression, we use a least fixed point operation, expressible in MSO logic.

Lemma 4.4.1 *Every routing expression R can be translated into an MSO formula $tr(R)(v_1, v_2)$ with two (first-order) free variables v_1 and v_2 such that for every graph S and nodes $a, b \in S$, there is an R -path from a to b if and only if $S, a, b \models tr(R)(v_1, v_2)$.*

Sketch of Proof: For atomic regular expressions and concatenation, we define $tr(R)(v_1, v_2)$ as follows:

$$tr(R)(v_1, v_2) \stackrel{\text{def}}{=} \begin{cases} f(v_1, v_2) & \text{if } R \text{ is } \underline{f} \\ f(v_2, v_1) & \text{if } R \text{ is } \overleftarrow{f} \\ \neg(c = v_1) \wedge (v_1 = v_2) & \text{if } R \text{ is } \neg c \\ u(v_1) \wedge (v_1 = v_2) & \text{if } R \text{ is } u \\ \neg u(v_1) \wedge (v_1 = v_2) & \text{if } R \text{ is } \neg u \end{cases}$$

$$tr(R_1.R_2)(v_1, v_2) \stackrel{\text{def}}{=} \exists v_3. tr(R_1)(v_1, v_3) \wedge tr(R_2)(v_3, v_2)$$

The formula $tr(R^*)(v_1, v_2)$ holds when the minimal set Y that contains v_1 and is closed under R , contains v_2 . Formally, we define

$$tr(R^*)(v_1, v_2) \stackrel{\text{def}}{=} \exists Y. (v_2 \in Y) \wedge Q(v_1, Y) \wedge \forall Y'. Q(v_1, Y') \Rightarrow Y \subseteq Y'$$

where $Q(v_1, Z)$ is $(v_1 \in Z) \wedge \forall v'_1, v'_2. (v'_1 \in Z) \wedge \varphi_R(v'_1, v'_2) \Rightarrow (v'_2 \in Z)$.

For example, the routing expression $R \stackrel{\text{def}}{=}} (\underline{n}, \neg y)^*$ is translated into the MSO formula $tr(R)(x, v) \stackrel{\text{def}}{=} \exists Y. (v \in Y) \wedge Q(x, Y) \wedge \forall Y'. Q(x, Y') \Rightarrow Y \subseteq Y'$, where $Q(x, Z)$ is $(x \in Z) \wedge \forall v'_1, v'_2. (v'_1 \in Z) \wedge \exists v'_3. (f(v'_1, v'_3) \wedge \neg(x = v'_3) \wedge (v'_3 = v'_2)) \Rightarrow (v'_2 \in Z)$.

Using the translation of regular expressions as defined above, it is easy to translate a general \mathcal{L}_0 formula to an *equivalent* MSO formula. For $\varphi \in \mathcal{L}_0$ over τ , $TR_2(\varphi)$ is an MSO formula over the same vocabulary τ . The translation TR_2 is defined inductively:

$$\begin{aligned} TR_2(c[R]p) &\stackrel{\text{def}}{=} \forall v_0, v_1, \dots, v_n. \varphi_R(c, v_0) \Rightarrow p(v_0, \dots, v_n) \\ TR_2(\varphi_1 \wedge \varphi_2) &\stackrel{\text{def}}{=} TR_2(\varphi_1) \wedge TR_2(\varphi_2) \\ TR_2(\neg\varphi_1) &\stackrel{\text{def}}{=} \neg TR_2(\varphi_1) \end{aligned}$$

For example, the \mathcal{L}_0 formula $\varphi \stackrel{\text{def}}{=}} x \langle \underline{n}, * \rangle y \wedge x [(\underline{n}, \neg y)^*] inv_{n, n'}$ which is part of a loop invariant of the reverse procedure (Section 4.3.3), is translated into the MSO formula

$$TR_2(\varphi) = tr(\underline{n}, *) (x, y) \wedge \forall v_0, v_1. tr((\underline{n}, \neg y)^*) (x, v_0) \Rightarrow (n(v_0, v_1) \Rightarrow n'(v_1, v_0))$$

where $tr(\underline{n}, *)$ and $tr((\underline{n}, \neg y)^*)$ are defined as above.

Lemma 4.4.2 *For all $\varphi \in \mathcal{L}_0$ and all graphs S , $S \models \varphi$ iff $S \models TR_2(\varphi)$.*

4.4.2 Decidability of MSO on Ayah Graphs

We define a set T^k of undirected graphs, each of which is a tree³ with at most k extra edges.

Definition 4.4.3 *An undirected graph B is in T^k if removing self loops and at most k additional edges from B results in an acyclic (undirected) graph.*

For a directed graph we define the corresponding undirected graph:

Definition 4.4.4 *Let $\mathcal{G}(S)$ denote the **Gaifman graph** of the graph S , i.e., an undirected graph obtained from S by removing node labels, edge labels, and edge directions (and parallel edges).*

We define a notion of simple tree-like (directed) graphs, called *Ayah graphs*.

Definition 4.4.5 (Ayah Graphs) *For $k \geq 0$, an Ayah graph of k is a graph S whose Gaifman graph is in T^k : $\mathcal{A}_k = \{S \mid \mathcal{G}(S) \in T^k\}$.*

Examples of graphs in \mathcal{A}_0 , \mathcal{A}_1 , and \mathcal{A}_2 are shown in Fig. 4.7. For $j = \{0, 1, 2\}$, a structure $S_j \in \mathcal{A}_j$ is shown in the left column, and the corresponding Gaifman graph $\mathcal{G}(S_j) \in T^j$ is shown in the right column; with j dashed edges. Removing the dashed edges from $\mathcal{G}(S_j)$ yields a tree.

³In this chapter, we use the term “tree” instead of the term “forest” to refer to an acyclic graph, possibly undirected.

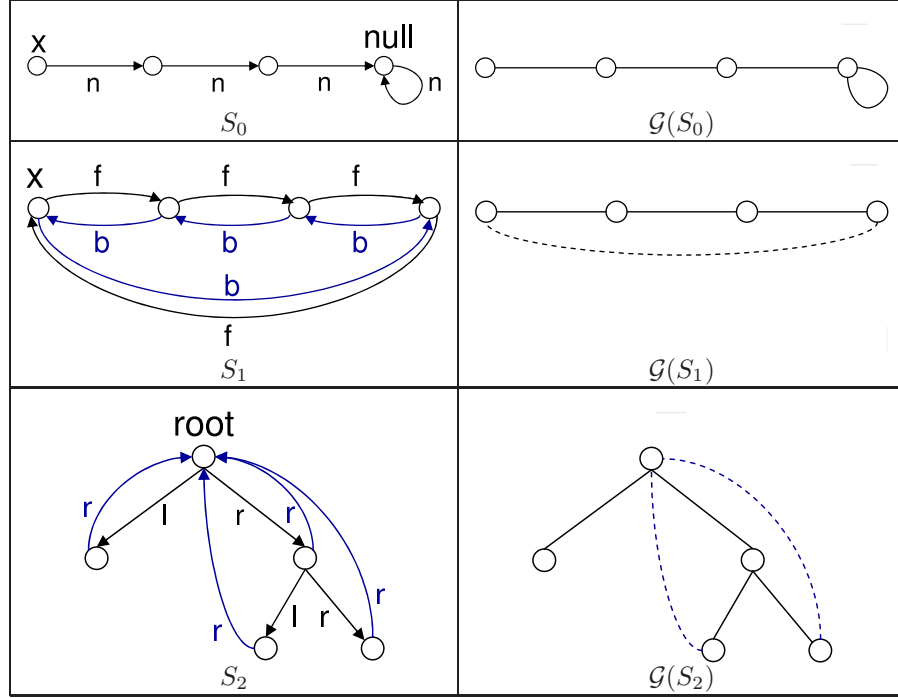


Figure 4.7: Examples of graphs in \mathcal{A}_0 , \mathcal{A}_1 , and \mathcal{A}_2 . For $j \in \{0, 1, 2\}$, $S_j \in \mathcal{A}_j$ (left column) and $\mathcal{G}(S_j) \in T^j$ (right column). Dashed edges denote extra edges removing which results in a tree.

The graph S_0 describes an acyclic singly-linked list pointed-to by x . The node labeled with *null* does *not* represent an element of the list: it is a “sink” node which models the *null* value, as explained in Section 4.3.2. In $\mathcal{G}(S_0)$, the self-loop is not dotted because Definition 4.4.3 ignores self-loops. (As we show later, self-loops can be easily handled, while larger cycles require a more complex treatment.) The graph S_1 describes a cyclic doubly-linked list. In $\mathcal{G}(S_1)$, a single edge represents the parallel edges of S_1 with different directions and different labels. The graph S_2 describes a tree with pointers from every tree node to the root. In $\mathcal{G}(S_2)$, removing a single edge cannot break both cycles, thus the graph S_2 is in \mathcal{A}_2 , but not in \mathcal{A}_1 .

Remark. For every graph S in \mathcal{A}_k , the tree width [RS86, Die00] of $\mathcal{G}(S)$ is at most $k + 1$, but can it can be strictly less than that. For example, a graph which consists of 17 simple disjoint cycles is in \mathcal{A}_{17} , but its tree width is 2.

The satisfiability problem of MSO logic on Ayah graphs can be reduced to the satisfiability problem of MSO logic on trees. The latter is decidable, due to the classical result by Rabin [Rab69]. This reduction provides a constructive way to check satisfiability of \mathcal{L}_1 formulas, using an existing decision procedure for MSO on trees, MONA [HJJ⁺95].

The reduction consists of two satisfiability-preserving translations: The first is a translation TR_3 from MSO on Ayah graphs to MSO on Σ -labeled trees, defined below. The second is a translation TR_4 from MSO on Σ -labeled trees to MSO on (infinite) binary trees.

Lemma 4.4.6 *There are translations TR_3 and TR_4 between MSO-formulas such that for every MSO-formula φ , there exists a graph $S \in \mathcal{A}_k$ that satisfies φ if and only if there exists a binary tree S' such that $S' \models (TR_3 \circ TR_4)(\varphi)$.*

We describe here only the translation TR_3 , and omit the (standard) translation, TR_4 .

Encoding \mathcal{A}_k Graphs as Σ -Labeled Trees

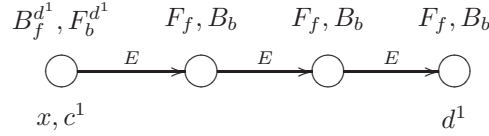
Given the vocabulary $\tau = \langle C, U, F \rangle$ and a number k we define a new vocabulary $\tau' = \langle C', U', \{E\} \rangle$, where E is the only binary relation, $C' = C \cup \{c^1, \dots, c^k\} \cup \{d^1, \dots, d^k\}$, and $U' = \{F_f, B_f, L_f, F_f^{d^i}, B_f^{d^i} \mid f \in F, i = 1, \dots, k\}$.

Let $\Sigma = \mathcal{P}(C' \cup U')$ be the set of all possible node labels from τ' . A Σ -labeled tree is a graph S over τ' that satisfies the following:

1. The E -edges form a directed forest: each node in S has at most one incoming E edge. An E -edge from node u_1 to node u_2 means that u_2 is a child of u_1 in the tree.
2. If a node has no incoming E -edge, then it must not be labeled by F_f, B_f , for any $f \in F$.

We use T_Σ to denote the set of all Σ -labeled trees.

Every graph in \mathcal{A}_k can be represented by a Σ -labeled tree. For example, consider the cyclic doubly-linked list S_1 from Fig. 4.7, defined over the vocabulary τ with $C = \{x\}$, $U = \{\}$, and $F = \{f, b\}$. The new vocabulary τ' consists of $C' = \{x, c^1, d^1\}$, $U' = \{F_f, F_b, F_f^{d^1}, F_b^{d^1}, B_f^{d^1}, B_b^{d^1}\}$, and $F' = \{E\}$. The graph S_1 can be represented by the following Σ -labeled tree (actually, it is a list in this example):



The graph S represented by a Σ -labeled tree has the same set of nodes as the tree. The labels of S are defined as follows. A graph node is labeled with the constants and unary relation symbols that hold for the corresponding node in the tree. An edge in the tree from node v to v' represents edges between the corresponding nodes v and v' in the graph. Additional labels on tree nodes represent the direction and the labels of the graph edges adjacent to the corresponding nodes in the graph, as follows.

For each binary relation symbol $f \in F$, we introduce two unary relation symbols F_f and B_f , denoting forward and backward f -edge. If there is an edge from v to v' in the tree, and v' is labeled with F_f in the tree, then there is an f -edge from v to v' in S . Similarly, if there is an edge from v' to v in the tree, and v is labeled with B_f in the tree, then there is an f -edge from v to v' in S . There is a self-loop of f on a node v in S if the node v in the tree is labeled with L_f . Also, each of the k pairs of constants c^i and d^i in a tree represents edges between the nodes corresponding to c^i and d^i in the graph. If v is labeled with c^i and $F_f^{d^i}$ in the tree, then there is an f -edge from v to the node labeled with d^i in S . If v is labeled with c^i and $B_f^{d^i}$ in the tree, then there is an f -edge from the node labeled with d^i to v in S .

For an MSO formula φ over τ , $TR_3(\varphi)$ is an MSO formula over the vocabulary τ' . The translation TR_3 is defined inductively on φ , where the only interesting part is the translation of a binary relation formula $f \in F$:

$$\begin{aligned} TR_3(f(v_1, v_2)) = & (E(v_1, v_2) \wedge F_f(v_2)) \\ & \vee (E(v_2, v_1) \wedge B_f(v_1)) \\ & \vee (E(v_1, v_2) \wedge v_1 = v_2 \wedge L_f(v_1)) \\ & \vee_{i=1}^k ((c^i = v_1 \wedge d^i = v_2 \wedge F_f^{d^i}(v_1)) \vee (c^i = v_2 \wedge d^i = v_1 \wedge B_f^{d^i}(v_2))) \end{aligned}$$

Lemma 4.4.7 *Let φ be an MSO formula. There is a graph $S \in \mathcal{A}_k$ such that $S \models \varphi$ if and only if there is a Σ -labeled tree $T \in T_\Sigma$ such that $T \models TR_3(\varphi)$.*

Proof: Given a graph S in \mathcal{A}_k , we can encode it as a Σ -labeled tree T as follows. First, remove all self loops and at most k additional edges from the Gaifman graph of S to obtain an acyclic undirected graph, U . It is easy to transform the undirected graph U into a directed forest T , by choosing one node in every connected component of U as a root, and directing all edges from it downwards. Then, we can set the labels of T uniquely from the labels of the corresponding nodes in S . To encode that an edge in S is labeled with f , we identify the corresponding edge in T , and label the target of the edge with a unary relation to remember the label f .

Given $T \in T_\Sigma$, we can uniquely reconstruct the graph $S \in \mathcal{A}_k$ that corresponds to it. Every node in T that is labeled with F_f has exactly one incoming edge, which defines the corresponding edge in S , labeled with f . For each $F_f^{d^i}$, at most one edge can be created in S , because TR_3 guarantees that in T the source is labeled with c^i , and the target is labeled with d^i , which are constants.

Theorem 4.4.8 *The satisfiability problem of MSO formulas is decidable on \mathcal{A}_k .*

Proof: Follows from Lemma 4.4.6 and [Rab69].

4.4.3 Normal Form of \mathcal{L}_0 Formulas

We define a normal-form formula to be a disjunction of conjunctions of formulas of the form $c\langle R \rangle c'$ and $c[R]p$.

Definition 4.4.9 (Normal-form formulas) *A formula in CL_0 is of the form*

$$\bigwedge_i \neg(c_i[R_i] \neg c'_i) \wedge \bigwedge_j c_j[R_j]p_j$$

A normal-form formula is a disjunction of CL_0 formulas.

A formula φ is in CL_1 if and only if $\varphi \in CL_0$ and $\varphi \in \mathcal{L}_1$, i.e., all the patterns that appear in φ satisfy the requirement of Definition 4.3.1.

For a formula $\varphi \in CL_0$, we use φ_\diamond to denote the first part of φ , namely $\bigwedge_i \neg c_i[R_i] \neg c'_i$, and φ_\square to denote the second part of φ , namely $\bigwedge_j c_j[R_j]p_j$. We use $|\varphi_\diamond|$ to denote the number of conjuncts in the formula φ_\diamond .

Note that while \mathcal{L}_0 is closed under negation, CL_0 is not. The following theorem shows that every \mathcal{L}_0 -formula can be effectively translated into an equi-satisfiable normal-form formula. The main difficulty is to translate a formula of the form $\neg c[R]p$, where p is an arbitrary pattern, into a formula in which negation appears only in front of constraints of the form $c'[R] \neg c''$.

Definition 4.4.10 *Let θ be the formula $\neg c[R]p$ over τ , where $p(v_0) = N(v_0, \dots, v_n) \Rightarrow \psi(v_0, \dots, v_n)$. We introduce new constant symbols c_0, \dots, c_n , and define $\tau' = \tau \cup \{c_0, \dots, c_n\}$. We define $tr(\theta)$ as follows:*

- *Translate $\neg\psi$ into an equivalent negated normal form formula ψ' ,*
- *Let θ' be $c\langle R \rangle c_0 \wedge N(c_0, \dots, c_n) \wedge \psi'(c_0, \dots, c_n)$, where every edge formula $v_i \xrightarrow{f} v_j$ that appears in N or ψ' is replaced by $c_i \langle \xrightarrow{f} \rangle c_j$.⁴*
- *If $\neg c\langle R \rangle c'$ appears in θ' , replace it with $c[R] \neg c'$, to obtain θ'' .*
- *Transform θ'' into an equivalent disjunctive normal form formula θ''' .*
- *Let $tr(\theta)$ be θ''' .*

The formula $tr(\theta)$ is a normal-form formula by Definition 4.4.9, because it is a disjunction of CL_0 -formulas. In fact, $tr(\theta)$ is a very simple formula: all the patterns in it are of the form $true \Rightarrow c \neq v_0$. Thus, negation can appear only in front of reachability constraints of the form $c[R] \neg c'$ where R does not contain the Kleene star operator.

Lemma 4.4.11 *For a graph S over τ , if S satisfies θ , then there exists an expansion of S to τ' , that satisfies $tr(\theta)$. For a graph S' over τ' , if $S' \models tr(\theta)$ then the restriction S of S' to τ satisfies φ .*

Theorem 4.4.12 *There is a computable translation TR_1 from \mathcal{L}_0 to a disjunction of formulas in CL_0 that preserves satisfiability.*

Sketch of Proof: For every formula $\varphi \in \mathcal{L}_0$ over τ , the formula $TR_1(\varphi)$ is a disjunction of formulas in CL_0 over τ' such that φ is satisfiable if and only if $TR_1(\varphi)$ is satisfiable. The vocabulary τ' is an extension of τ with new constant symbols. The translation $TR_1(\varphi)$ is defined as follows:

1. Translate φ into an equivalent formula φ' in negated normal form using deMorgan rules to push negations inwards.
2. Replace every sub-formula $\neg c[R]p$ that appears in φ' with $tr(\neg c[R]p)$, as in Definition 4.4.10. The resulting formula φ'' is satisfiable if and only if φ' is satisfiable, by Lemma 4.4.11. Note that this translation only preserves satisfiability (not equivalence).
3. Translate φ'' into an equivalent disjunctive normal form formula φ''' . All atomic formulas are of the form $c[R] \neg c'$.

The result of $TR_1(\varphi)$ is φ''' .

⁴Recall from Section 4.1.1 that $c\langle R \rangle c'$ is a shorthand for $\neg c[R] \neg c'$.

The translation is applicable to the full \mathcal{L}_0 logic, in which case the reachability constraints in φ_{\square} can contain arbitrary patterns.

The translation TR_1 may introduce only patterns of the form $true \Rightarrow c_2 \neq v_0$ beyond those patterns that appear in the input formula. This observation yields the following corollary:

Corollary 4.4.13 *For $\varphi \in \mathcal{L}_1$, the translation TR_1 returns a disjunction of formulas in CC_1 (and preserves satisfiability).*

4.4.4 Decidability of \mathcal{L}_1

The following theorem states that CC_1 has an Ayah-model property, i.e., every satisfiable CC_1 formula φ has a model in $\mathcal{A}_{f(\varphi)}$ where $f(\varphi)$ is defined by

$$f(\varphi) \stackrel{\text{def}}{=} 2 \times n \times |C| \times |\varphi_{\diamond}| \quad (4.4)$$

Here, we assume that for every routing expression that appears in φ_{\diamond} there is an equivalent automaton with at most n states.

Theorem 4.4.14 (Ayah model property of \mathcal{L}_1) *If $\varphi \in CC_1$ is satisfiable, then φ is satisfiable by a graph in $\mathcal{A}_{f(\varphi)}$, where f is defined in (4.4).*

A non-trivial proof of this theorem is presented in Section 4.5.

Theorem 4.4.15 *The satisfiability problem of \mathcal{L}_1 is decidable.*

Proof: Follows from combining the results of Theorem 4.4.12, Theorem 4.4.14, Lemma 4.4.2, Theorem 4.4.8.

4.5 Ayah Model Property of \mathcal{L}_1

In this section we provide a detailed proof of the main technical theorem of this chapter, Theorem 4.4.14. Before diving into the details, we explain the main proof at a high-level.

Given a normal-form formula $\varphi \in CC_1$ and a graph S such that $S \models \varphi$, we construct a graph S' and show that $S' \models \varphi$ and $S' \in \mathcal{A}_k$.

The construction operates as follows. We construct a pre-model S_0 of S and φ , which satisfies all constraints of the form $c\langle R \rangle c'$ in φ . The idea is to extract from S a witness path for each constraint of the form $c\langle R \rangle c'$ in φ , and define S_0 to be the union of these witness paths (Section 4.5.5).

The pre-model S_0 may violate some of the constraints of the form $c[R]p$ in φ . Consider the case when the pattern p contains a positive occurrence of edge formula or equality formula. If a graph G violates a constraint $c[R]p$, then there is an enabled merge operation or edge-addition operation, depending on the pattern p (Section 4.5.3).

For example, if p is of the form $N(v_0, v_1, v_2) \Rightarrow v_1 = v_2$, it defines a merge operation. We say that this merge operation is enabled in a graph G (by $c[R]p$) when G contains a node w_0 reachable by an R -path from c and *distinct* nodes w_1 and w_2 forming the neighborhood $N(w_0, w_1, w_2)$. Applying this operation means merging the nodes w_1 and w_2 . After merging w_1 and w_2 , other merge operations may still be enabled in G by $c[R]p$. If there are no more enabled operations in G , then $G \models c[R]p$. Similarly, if p is of the form $N(v_0, v_1, v_2) \Rightarrow v_1 \xrightarrow{f} v_2$, it defines an edge-addition operation. Applying this operation means adding an f -edge.

Given a pre-model S_0 , we apply all enabled operations in any order, producing a sequence of distinct graphs S_0, S_1, \dots until the last graph S' has no enabled operations. Thus, S' satisfies all constraints of the form $c[R]p$ where p contains a positive occurrence of edge formula or equality formula. We show that applying any enabled operation preserves witness paths for the constraints of the form $c\langle R \rangle c'$. Thus, S' also satisfies all constraints of the form $c\langle R \rangle c'$. This construction also guarantees that S' satisfies all the constraints of the form $c[R]p$ where p is a negative formula. To show this formally, we use homomorphisms (Section 4.5.4) which preserves existence of edges and both existence and absence of labels on nodes (preserving absence of labels is non-standard).

Finally, the fact that S' is in \mathcal{A}_k is proved by induction. By construction, S_0 is in \mathcal{A}_k (Lemma 4.5.11), and \mathcal{A}_k is closed under operations enabled by \mathcal{L}_1 formulas (Lemma 4.5.5). The proof of closure properties of \mathcal{A}_k is based on closure properties for a class of undirected graphs, T^k (Lemma 4.5.1).

The rest of the section describes the building blocks of the proof of Theorem 4.4.14: closure properties of T^k (Section 4.5.1), closure properties of \mathcal{A}_k (Section 4.5.2), the definition of operations enabled by \mathcal{L}_1 formulas (Section 4.5.3), the definition of homomorphism relation and its properties (Section 4.5.4), and the definition of witness splitting and properties of a pre-model (Section 4.5.5). The proof of Theorem 4.4.14 concludes the section.

4.5.1 Trees with Extra Edges

Recall from Definition 4.4.3 that T^k is a set of undirected graphs that are trees with k extra edges. In this section we prove that T^k is closed under merging of vertices at distance at most 2.

The *distance* between the vertices v_1 and v_2 in an undirected graph B is the number of edges on the shortest path between v_1 and v_2 in B .

Merging two vertices in an undirected graph is defined in the usual way, by gluing these vertices. Formally, let the undirected graph B' denote the result of merging nodes v_1 and v_2 in B . The set of vertices of B' is $V^{B'} \stackrel{\text{def}}{=} (V^B \setminus \{v_1, v_2\}) \cup \{v_{12}\}$, where v_{12} is a new vertex. Let $m: V^B \rightarrow V^{B'}$ be defined as follows:

$$m(v) = \begin{cases} v_{12} & \text{if } v = v_1 \text{ or } v = v_2 \\ v & \text{otherwise} \end{cases}$$

If there is an edge e between the vertices v_1 and v_2 in B then there is an edge $m(e)$ between $m(v_1)$ and $m(v_2)$ in B' . If there is an edge e between v'_1 and v'_2 in B' then there exist vertices v_1 and v_2 in B such that $m(v_1) = v'_1$, $m(v_2) = v'_2$, and there is an edge between v_1 and v_2 in B .

Lemma 4.5.1 *Assume that B is in T^k and vertices v_1 and v_2 are at distance at most two in B . The graph B' obtained from B by merging v_1 and v_2 in B is also in T^k .*

Proof: By definition of T^k , there exists a set of edges $D \subseteq E$ such that $B \setminus D$, denoted by T , is acyclic and $|D| \leq k$. We show how to transform D into $D' \subseteq E'$ such that $B' \setminus D'$, denoted by T' , is acyclic and $|D'| \leq k$. We consider only the case when v_1 and v_2 are at distance of exactly two in B , i.e., there is a vertex v_0 distinct from v_1 and v_2 , an edge e_1 between v_1 and v_0 , and an edge e_2 between v_0 and v_2 . We consider three cases, depicted in Fig. 4.8.

- If $e_1, e_2 \notin D$, let $D' = \{m(e) | e \in D\}$.
- Assume that $e_1 \notin D$ and $e_2 \in D$. If v_2 is not reachable from v_1 in T , let $D' = \{m(e) | e \in D\}$, thus $|D'| \leq k$.

If v_2 is reachable from v_1 in T , there is at most one (simple) path from v_1 to v_2 in T , because T is acyclic. If the path contains e_1 , we define D' as before: $D' = \{m(e) | e \in D\}$.

If the path from v_1 to v_2 does not contain e_1 , let e_3 be the first edge on the path from v_1 to v_2 (see the second case in Fig. 4.8).⁵ To obtain D' from D , we remove e_2 and add e_3 : $D' = (\{m(e) | e \in D\} \setminus \{m(e_2)\}) \cup \{m(e_3)\}$. The size of D' is the same as the size of D , because $e_2 \in D$.

- Assume that $e_1, e_2 \in D$. If v_2 is not reachable from v_1 , we can use the simple construction $D' = \{m(e) | e \in D\}$. It follows that $|D'| = |D| - 1$, because both e_1 and e_2 are mapped to the same edge $e' = m(e_1) = m(e_2)$, and no multiple edges are allowed.

If v_2 is reachable from v_1 , let e_3 be the first edge on the path. We define $D' = \{m(e) | e \in D\} \cup \{m(e_3)\}$ (see the third case in Fig. 4.8). Same construction applies when v_1 or v_2 are reachable from v_0 .

⁵Note that we cannot use the simple D' definition as before, because merging v_1 and v_2 in T to obtain T' creates a cycle that does not involve e_1 . We observe that, in this case, the subgraph reachable from v_1 through e_1 in T remains acyclic after the merge operation, because it is disjoint from the subtree of v_2 . Thus, e_1 need not be removed from T .

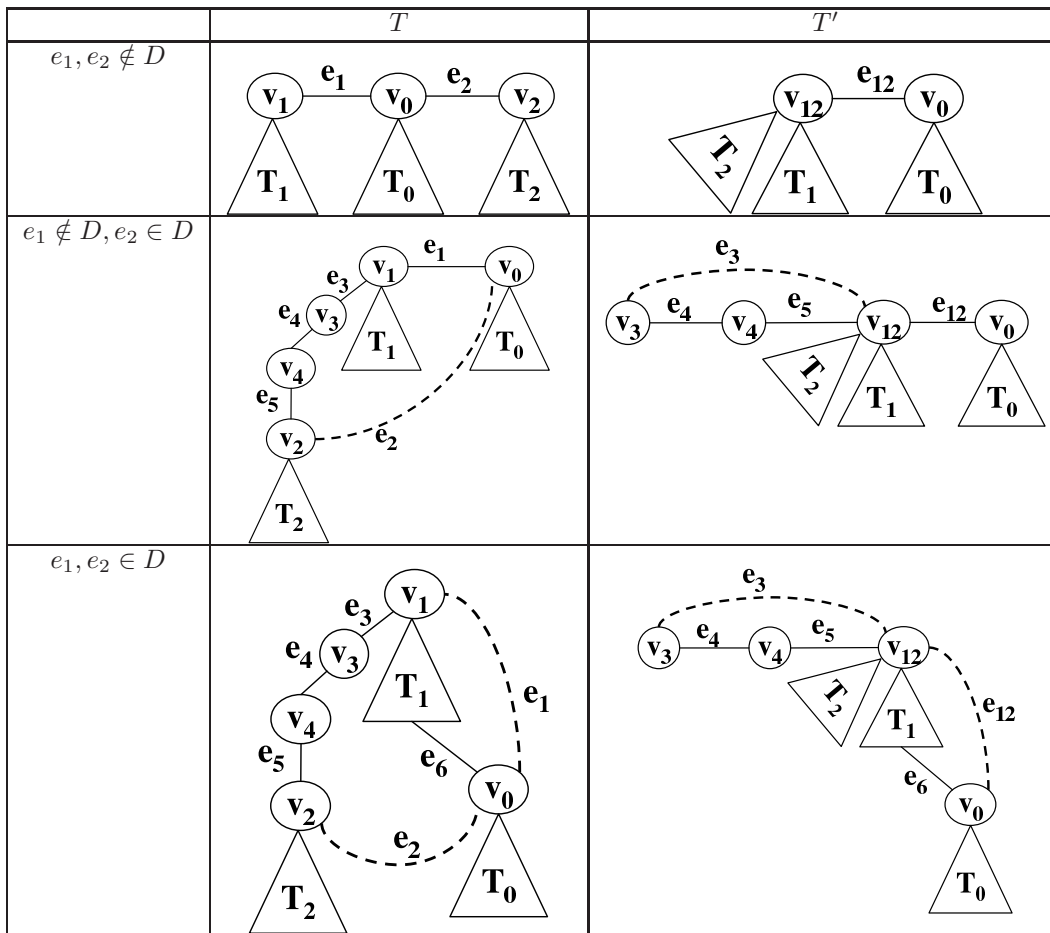


Figure 4.8: Merge operation on T^k -graphs. Dotted lines represent additional edges, i.e., edges of a T^k -graph that do not belong to the tree. The vertex v_{12} and the edge e_{12} in T' result from merging the vertices v_1 and v_2 , and the edges e_1 and e_2 in T .

4.5.2 Ayah Graphs

In this section we prove that \mathcal{A}_k is closed under edge-addition operations at distance at most one (Lemma 4.5.2), and under merge operations at distance at most 2 (Lemma 4.5.3).

The *distance* between nodes v_1 and v_2 in a graph S is the distance between v_1 and v_2 in $\mathcal{G}(S)$, i.e., the number of edges on the shortest path between v_1 and v_2 in $\mathcal{G}(S)$.

It is easy to see that \mathcal{A}_k is closed under edge-addition operations at distance at most one, which means adding an edge in parallel to an existing one (distance one) or adding a self-loop (distance zero).

Lemma 4.5.2 (Adding edges at distance ≤ 1 in \mathcal{A}_k) *Assume that the graph S' is obtained from S by adding an edge from v_1 to v_2 in S . If S is in \mathcal{A}_k and nodes v_1 and v_2 are at distance at most 1 in S , then S' is in \mathcal{A}_k .*

Proof: Distance at most 1 between v_1 and v_2 means that there is already an edge between v_1 and v_2 . Addition of edges to S in parallel to existing edges does not affect the $\mathcal{G}(S)$, and self-loops do not affect T^k .

Merging two nodes in a graph is defined in the usual way by gluing these nodes. Formally, let S' be the result of merging the nodes v_1 and v_2 in S . The set of nodes of S' is $V^{S'} \stackrel{\text{def}}{=} (V^S \setminus \{v_1, v_2\}) \cup \{v_{12}\}$, where v_{12} is a new node. We define $m: V^S \rightarrow V^{S'}$ as follows:

$$m(v) = \begin{cases} v_{12} & \text{if } v = v_1 \text{ or } v = v_2 \\ v & \text{otherwise} \end{cases}$$

The interpretation of constant and relation symbols in S' is defined as follows:

1. For every constant symbol $c \in \tau$, and for every node $v \in S$, v is labeled with c in S if and only if $m(v)$ is labeled with c in S' .
2. For every unary relation symbol $\sigma \in \tau$, and for every node $v \in S$, if v is labeled with σ in S then $m(v)$ is labeled with σ in S' .
3. For every unary relation symbol $\sigma \in \tau$, and for every node $v' \in S'$, if v' is labeled with σ in S' then there exists a node v in S such that $m(v) = v'$ and v is labeled with σ in S .
4. For every binary relation symbol $\sigma \in \tau$, and every pair of nodes $w_1, w_2 \in S$, if there is an edge from w_1 to w_2 labeled with σ then there is an edge from $m(w_1)$ to $m(w_2)$ in S' labeled with σ .
5. for every binary relation symbol $\sigma \in \tau$, and every pair of nodes $w'_1, w'_2 \in S'$, if there is an edge from w'_1 to w'_2 labeled with σ in S' then there are nodes w_1 and w_2 in S such that $m(w_1) = w'_1$, $m(w_2) = w'_2$, and there is an edge from w_1 to w_2 in S labeled with σ .

Later, we guarantee that merge operations are applied only to those nodes which are labeled by the same unary relations and constants.

The proof that \mathcal{A}_k is closed under merge operations at distance at most two is based on the result of Lemma 4.5.1 from the previous section.

Lemma 4.5.3 (Merging nodes at distance ≤ 2 in \mathcal{A}_k) *Assume that the graph S' is obtained from S by merging v_1 and v_2 in S . If S is in \mathcal{A}_k and nodes v_1 and v_2 are at distance at most 2 in S , then S' is in \mathcal{A}_k .*

Proof: To show that $S' \in \mathcal{A}_k$, it is sufficient to show that $\mathcal{G}(S') \in T^k$. We use the definitions of a Gaifman graph and a merging operation. First, merging the nodes of $\mathcal{G}(S)$ that correspond to v_1 and v_2 in $\mathcal{G}(S)$, results in $\mathcal{G}(S')$. Second, the distance between v_1 and v_2 in $\mathcal{G}(S)$ is at most 2 because the distance between the corresponding nodes in S is at most 2. Third, $\mathcal{G}(S) \in T^k$, because $S \in \mathcal{A}_k$. Thus, using Lemma 4.5.1, we get that $\mathcal{G}(S') \in T^k$.

4.5.3 Graph Operations Enabled by \mathcal{L}_1 Formulas

The notion of enabled operations defined in this section is used for defining the construction in the proof of Theorem 4.4.14.

Let $p(v_0) \stackrel{\text{def}}{=} N(v_0, \dots, v_n) \Rightarrow \psi(v_0, \dots, v_n)$ be an \mathcal{L}_1 pattern. Let S be a graph, and w_1, w_2 nodes in S .

We say that *merge operation of w_1 and w_2 is enabled* (by $c[R]p$) when (a) the equality between variables ($v_1 = v_2$) appears positively in ψ , (b) we can assign nodes w_0, \dots, w_n to v_0, \dots, v_n , respectively, such that there

is an R -path from c to w_0 , $N(w_0, \dots, w_n)$ holds but $\psi(w_0, \dots, w_n)$ does not hold, and (b) w_1 and w_2 are **distinct** nodes. Merging the nodes w_1 and w_2 disables this merge operation (other merge operations may still be enabled after merging w_1 and w_2).

We say that *edge-addition between w_1 and w_2 is enabled* (by $c[R]p$) when (a) the edge formula $(v_1 \xrightarrow{f} v_2)$ appears positively in ψ , (b) we can assign nodes w_0, \dots, w_n to v_0, \dots, v_n , respectively, such that there is an R -path from c to w_0 , $N(w_0, \dots, w_n)$ holds but $\psi(w_0, \dots, w_n)$ does not hold, and (c) there is **no** f -edge from w_1 to w_2 . We can add an f -edge from w_1 and w_2 to discharge this assignment.

Lemma 4.5.4 *Let $N(v_0, \dots, v_n)$ be a neighborhood formula, and S be a graph with an assignment to v_0, \dots, v_n that satisfies N . If the variables v_1 and v_2 are at distance at most k in N , then the nodes assigned to v_1 and v_2 are at distance at most k in S .*

Proof: Follows from the definition of neighborhood as a conjunction of edges (Definition 4.1.2).

The following lemma is the key observation of the proof.

Lemma 4.5.5 *Let $p(v_0) \stackrel{\text{def}}{=} N(v_0, v_1, \dots, v_n) \Rightarrow \psi(v_0, \dots, v_n)$ be an \mathcal{L}_1 pattern. Let S be a graph, and w_1, w_2 nodes in S . Assume that a merge (an edge-addition) operation is enabled in a graph S between nodes w_1 and w_2 by a reachability constraint $c[R]p$. If $S \in \mathcal{A}_k$, then the result of merging (adding an edge) between w_1 and w_2 is a graph in \mathcal{A}_k .*

Proof: Suppose that a merge operation is enabled in S between nodes w_1 and w_2 . It is possible to assign nodes w_0, \dots, w_n to the variables v_0, \dots, v_n , such that N holds. In particular, w_1 is assigned to v_1 and w_2 is assigned to v_2 , and the equality $v_1 = v_2$ appears positively in ψ . According to the equality restriction on \mathcal{L}_1 patterns, v_1 and v_2 are at distance at most 2 in N . By Lemma 4.5.4, w_1 and w_2 are at distance at most 2 in S . Thus, by Lemma 4.5.3 we get that the result of merging w_1 and w_2 is a graph in \mathcal{A}_k , because S is in \mathcal{A}_k . The proof for edge-addition is similar, using Lemma 4.5.2.

4.5.4 Homomorphism Preservation

In this section, we give a slightly non-standard definition of homomorphism between graphs. It preserves existence of edges and both existence and absence of labels on nodes (preserving absence of labels is non-standard). The homomorphism relation is preserved by $C\mathcal{L}_1$ formulas, and also by merging operations.

Definition 4.5.6 (Homomorphism) *Let S_1 and S_2 be graphs over the same vocabulary τ . A homomorphism from S_1 to S_2 is a mapping $h: V^{S_1} \rightarrow V^{S_2}$ such that*

1. *for every constant symbol and unary relation symbol $\sigma \in \tau$, and for every $v \in S_1$, v is labeled with σ in S_1 if and only if $h(v)$ is labeled with σ in S_2 .*
2. *for every binary relation symbol $\sigma \in \tau$, and every pair of nodes $v_1, v_2 \in S_1$, if there is an edge from v_1 to v_2 in S_1 labeled with σ , then there is an edge from $h(v_1)$ to $h(v_2)$ in S_2 labeled with σ .*

Lemma 4.5.7 *Let $h: S_1 \rightarrow S_2$ be a homomorphism. If $S_1 \models c_1(R)c_2$ then $S_2 \models c_1(R)c_2$. Dually, if $S_2 \models c[R]p$, and p does not contain positive occurrences of edge formulas or equality formulas, then $S_1 \models c[R]p$.*

Sketch of Proof: If $S_1 \models c_1(R)c_2$, there exists an R -path from c_1 to c_2 . By definition of homomorphism from S_1 to S_2 , the same path exists in S_2 . Thus, $S_2 \models c_1(R)c_2$.

For the sake of contradiction, assume that $S_2 \models c[R]p$ but $S_1 \not\models c[R]p$. That is, there exists an R -path from c to some node v in S_1 and v does not satisfy the pattern p . The same path exists in S_2 , due to the homomorphism from S_1 to S_2 . To obtain a contradiction, we show that $h(v)$ does not satisfy the pattern p in S_2 . The formula p is of the form $N \Rightarrow \psi$, where N contains only positive occurrences of edge formulas. By assumption, we get that ψ does not contain positive occurrences of edge formulas or equality formulas. Thus, the formula p does not contain positive occurrences of edge formulas and equality formulas. If S_1 does not satisfy p , there exists a subgraph in S_2 which satisfies $\neg p$. This subgraph exists in S_2 as well, due to homomorphism.⁶ Thus, S_2 satisfies $\neg p$, and a contradiction is obtained.

Lemma 4.5.8 *Assume that f is a homomorphism from S_1 to S , and S_2 is obtained by merging the nodes v_1 and v_2 in S_1 . If $f(v_1) = f(v_2)$ then there is a homomorphism from S_2 to S .*

⁶Note that $\neg p$ may contain negative occurrences of unary formulas, but these are also preserved under the (non-standard) homomorphism relation we are using.

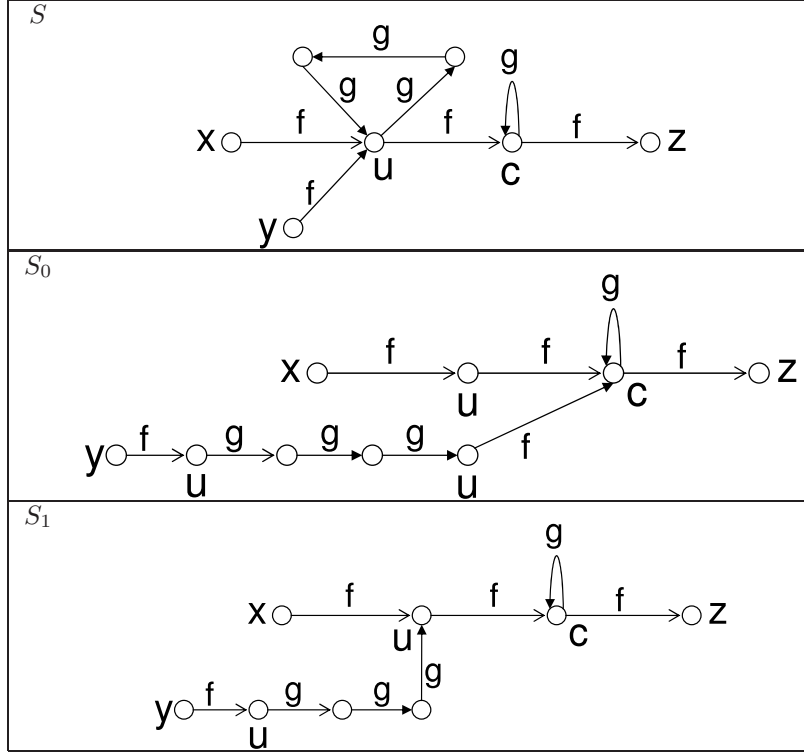


Figure 4.9: The graph S satisfies the formula in (4.5), and $S \in \mathcal{A}_1$. A pre-model of S is S_0 . Note that $S_0 \in \mathcal{A}_0$. The graph S_1 is the result of applying a merge operation to S_0 . Note that S_1 satisfies the formula in (4.5), and $S_1 \in \mathcal{A}_0$. The graph S_1 is the final result of the construction used in the proof of Theorem 4.4.14.

4.5.5 Witness Splitting

A witness W for $c_1 \langle R \rangle c_2$ in a graph S , is a path in S , labeled with a word $w \in L(R)$, from the node labeled with c_1 to the node labeled with c_2 . Note that the nodes and edges on a witness path for R need not be distinct. S contains a witness for $c_1 \langle R \rangle c_2$ if and only if $S \models c_1 \langle R \rangle c_2$.

Using a witness W for $c_1 \langle R \rangle c_2$ in S , we construct a graph W' that consists of a path, also labeled with w , that starts at the node labeled by c_1 and ends at the node labeled by c_2 . Intuitively, we create W' by duplicating a node of S each time the witness path W traverses it, unless the node is labeled with a constant. The nodes in W' are named $t_{v,l}$ where v is a node in S and $l \geq 0$ is an integer. For $l > 0$, a node $t_{v,l}$ in W' corresponds to the l -th occurrence of v on the witness path W , if a node v in S is not labeled with a constant. If v is labeled with a constant, we create for it a unique node $t_{v,0}$ in W' even if v is traversed several times by W . As a result, all shared nodes in W' are labeled with constants. Also, every cycle contains a node labeled with a constant. By construction, W' satisfies $c_1 \langle R \rangle c_2$.

For example, consider the formula

$$\varphi \stackrel{\text{def}}{=} x \langle \underline{f}^* \rangle z \wedge y \langle \underline{f}, (\underline{g}^+, (c|u), \underline{f})^* \rangle z \wedge c[\epsilon] \text{uns}_f \quad (4.5)$$

where u is a unary relation symbol and c is a constant symbol. Fig. 4.9 shows a graph S which satisfies φ . The shortest witness path for $x \langle \underline{f}^* \rangle z$ is labeled with the word $\underline{f}, \underline{f}, \underline{f}$. The shortest witness path for $y \langle \underline{f}, (\underline{g}^+, (c|u), \underline{f})^* \rangle z$ is labeled with the word $\underline{f}, \underline{g}, \underline{g}, \underline{g}, u, \underline{f}, \underline{g}, c, \underline{f}$. Note that this witness traverses each of the nodes labeled by u and by c twice. To split this witness, the node marked by u is duplicated, while the node marked by c is not duplicated, because c is a constant. After splitting the witnesses, we construct a pre-model of S , denoted by S_0 , by taking the union of both witness paths and merging the nodes of the different witness paths which are labeled with the same constant.

Formally, the witness path W is a sequence of nodes from S : t_1, t_2, \dots, t_r , where $t_i \in S$. Let $C(t_i)$ denote

the set of constant symbols that label the node t : $C(t_i) \stackrel{\text{def}}{=} \{\sigma \in C \mid C^S(\sigma) = t_i\}$. We define a mapping $d(t_i)$ as follows:

$$d(t_i) \stackrel{\text{def}}{=} \begin{cases} t_{v,0} & \text{if } C(t_i) \neq \emptyset \text{ and } t_i \text{ is the node } v \\ t_{v,l} & \text{if } t_i \text{ is the } l\text{-th occurrence of the node } v \in S \text{ on the path } W \end{cases}$$

W' is a graph with nodes $\{d(t_1), \dots, d(t_r)\}$. If the witness path W goes from t_i to t_{i+1} through an edge labeled with $f_i \in F$, then there is an edge in W' labeled with f_i from $d(t_i)$ to $d(t_{i+1})$. Note that W' contains only edges traversed by the witness path. For every unary relation and constant symbol $\sigma \in C \cup U$ and node $t_i \in W$, $d(t_i)$ is labeled with σ in W' if and only if t_i is labeled with σ in S .

We say that W' is the result of *splitting* the witness W . We say that W is the *shortest witness* for $c_1 \langle R \rangle c_2$ if any other witness path for $c_1 \langle R \rangle c_2$ is at least as long as W .

For a formula $\varphi \in \mathcal{CL}_1$ and a graph S such that $S \models \varphi$, we define a *pre-model of a S and φ* to be the graph S_0 constructed as follows.

- Let W_i denote a shortest witness in S for every $c_i \langle R \rangle c'_i$ in φ_\diamond .
- Let W'_i be the result of splitting the witness W_i . Let $t_{v,l}^i$ be the names the nodes of W'_i .
- Let S'_0 be a disjoint union of all W'_i 's.
- For every $c \in C$, if S'_0 does not contain any node labeled with c , add a new node $t_{v,0}^0$ to S'_0 , where v is the node in S labeled with c . For all $\sigma \in C \cup U$, $t_{v,0}^0$ is labeled with σ in S'_0 if and only if v is labeled with σ in S .
- The graph S_0 is the result of merging all nodes that are labeled with the same constants, i.e., nodes $t_{v,0}^i$ for all i are merged and the new node named $t_{v,0}^0$.

Note that S'_0 cannot be used as a legal interpretation for \mathcal{L}_0 formulas over τ , because it may contain several nodes labeled with the same constant, or no interpretation for some constants. These problems are addressed by the last two steps of the construction.

By construction, S_0 contains a witness for each $c_1 \langle R \rangle c_2$ in φ_\diamond .

Lemma 4.5.9 *If $S \models \varphi$ and S_0 is a pre-model of S and φ , then $S_0 \models \varphi_\diamond$.*

Lemma 4.5.10 *Let S_0 be a pre-model of S and φ . There is a homomorphism $h_0: S_0 \rightarrow S$ defined by $h_0(t_{v,l}^i) = v$.*

Proof: We define $h'_0: S'_0 \rightarrow S$ by $h'_0(t_{v,l}^i) = v$. The mapping h'_0 preserves existence of edges and the presence and absence of node labels between S'_0 and S because it is preserved for every W' separately, by definition of witness splitting, and S'_0 is a *disjoint* union of W'_i 's. Thus, h'_0 is a homomorphism.

Because S_0 is obtained from S'_0 by merging nodes that are mapped by h'_0 to the same node in S , the mapping h_0 is also a homomorphism, by Lemma 4.5.8.

Lemma 4.5.11 *For $\varphi \in \mathcal{CL}_1$, if S_0 is a pre-model of S and φ , then $S_0 \in \mathcal{A}_f(\varphi)$, where f is defined in (4.4).*

Proof:

Recall that for every routing expression that appears in φ_\diamond there is an equivalent automaton with at most n states. If a node is visited more than once in the same state of the automaton, the path can be shortened by removing the part traversed between the two visits. Thus, a shortest witness visits a node at most n times. In the worst case, each time a shortest witness visits a node, it enters and exits the node with a different edge. Because S_0 consists of $|\varphi_\diamond|$ shortest witnesses, there are at most $2 \times n \times |\varphi_\diamond|$ edges adjacent to any node.

In fact, by construction of S_0 , only nodes labeled by constants in S_0 can have more than two adjacent edges. Thus, every (simple) cycle in S_0 must go through a constant. To break all cycles in S_0 (and, thus, in its Gaifman graph), it is sufficient to remove all the edges adjacent to nodes labeled with constants, i.e., at most $k = 2 \times n \times |\varphi_\diamond| \times |C|$ edges. It follows that $S_0 \in \mathcal{A}_k$.⁷

⁷This bound is not tight.

4.5.6 \mathcal{A}_k -Model Property of \mathcal{L}_1

Theorem 4.4.14(Ayah model property of \mathcal{L}_1) *If $\varphi \in \mathcal{CL}_1$ is satisfiable, then φ is satisfiable by a graph in $\mathcal{A}_f(\varphi)$, where f is defined in (4.4).*

Proof: Given a graph S such that $S \models \varphi$, we construct a graph S' and show that $S' \in \mathcal{A}_k$ and $S' \models \varphi$.

First, we construct a pre-model S_0 of S and φ , and define the mapping $h_0: S_0 \rightarrow S$ according to Lemma 4.5.10. Then, we apply all enabled merge operations and all enabled edge-addition operations in any order, producing a sequence of distinct graphs S_0, S_1, \dots, S_r , until S_r has no enabled operations. The result $S' = S_r$.

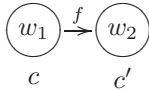
Formally, for every $c[R]p \in \varphi$ and ever pair of nodes $w_1, w_2 \in S_j$,

- If a merge operation is enabled, and $h_j(w_1) = h_j(w_2)$ in S_j then construct S_{j+1} by merging w_1 and w_2 , and define $h_{j+1}: S_{j+1} \rightarrow S$ to be $h_{j+1}(w) = h_j(w_1)$ if w is the result of merging w_1 and w_2 , otherwise $h_{j+1}(w) = h_j(w)$.
- If an edge-addition operation is enabled for $f \in F$, and there is an f -edge from $h_j(w_1)$ to $h_j(w_2)$ in S then construct S_{j+1} by adding an f -edge from w_1 to w_2 , and define $h_{j+1}: S_{j+1} \rightarrow S$ to be the same as h_j .

For example, the pre-model S_0 shown in Fig. 4.9 does not satisfy the constraint $c[\epsilon]uns_f$ from (4.5), which requires that the node labeled with c have at most one incoming f -edge. The result of applying the corresponding merge operation is the structure S_1 , also shown in Fig. 4.9.

An enabled merge operation is not applied to S_j if the corresponding nodes in the original model S are distinct. Similarly, an enabled edge-addition is not applied, unless the corresponding edge is present in S . This allows us to deal with disjunctions in patterns. For example,

$$\mathbf{let} \ p(v_0) \stackrel{\text{def}}{=} (v_0 \xrightarrow{f} v_1) \Rightarrow (v_0 = v_1 \vee (v_0 \xrightarrow{g} v_1) \vee (v_0 \xrightarrow{g'} v_1)) \ \mathbf{in} \\ c \langle \xrightarrow{f}^* \rangle c' \wedge c \langle \xrightarrow{f}^* \rangle p \wedge (c \neq c')$$

Suppose that S_0 looks like this:  The nodes w_1 and w_2 are labeled with the constants c and c' ,

respectively. Both merge and edge-addition operations are enabled in S_0 by $c \langle \xrightarrow{f}^* \rangle p$. Had we applied the merge operation, we would have immediately obtained a contradiction with $c \neq c'$. However, if we consult the original model, we find out that the corresponding nodes are distinct,⁸ but there is a g -edge between them. Therefore, adding a g -edge to S_0 would not lead to a contradiction.

Remark. Even when we consult with S whether to apply an enabled operation or not, we do not merge more than necessary, or add more edges than necessary. In the previous example, after adding g the formula holds, i.e., the edge-addition operation of g' is not enabled any more. However, a different order of application of the enable operations may produce different graphs at the end. Fortunately, it does not affect the size of \mathcal{A}_k , or the decidability.

The process described above terminates after a finite number of steps, because in each step either the number of nodes in the graph is decreased (by merge operations) or the number of edges is increased (by edge-addition operations). For a fixed vocabulary and a fixed number of nodes, the number of edges that can be added to the graph is bounded, because a pair of nodes in a graph can have at most one f edge in each direction, for every $f \in F$.

To show that $S' \in \mathcal{A}_k$, we prove a stronger claim that for all j , $S_j \in \mathcal{A}_k$. In particular, it follows that $S' \in \mathcal{A}_k$. Recall that all operations applied in the process above are enabled by \mathcal{L}_1 patterns. The key observation of the proof is that \mathcal{A}_k is closed under all operations enabled by \mathcal{L}_1 patterns (Lemma 4.5.5). This is the only place in our proof where we use the distance restriction of \mathcal{L}_1 patterns. The proof proceeds by induction on the process described above. Initially, S_0 is in \mathcal{A}_k , by Lemma 4.5.11. By inductive hypothesis, $S_j \in \mathcal{A}_k$. Because S_{j+1} is obtained from S_j by an operation that is enabled by an \mathcal{L}_1 pattern, we get that $S_{j+1} \in \mathcal{A}_k$, using Lemma 4.5.5.

To show that $S' \models \varphi$, we observe that the graphs generated by the process above are related to each other by different homomorphism relations (Definition 4.5.6), as depicted in Fig. 4.10.

First, each step of the process can be seen as a transformation t_j from S_{j-1} to S_j , which is defined by an operation applied at step j . That is, t_j is either a merge operation or an edge-addition operation. It is easy to see that both operations are homomorphisms. Therefore, each t_j is a homomorphism, for all j .

⁸The nodes $h_0(w_1)$ and $h_0(w_2)$ in S are distinct, because our construction of pre-model S_0 does not split nodes labeled by constants.

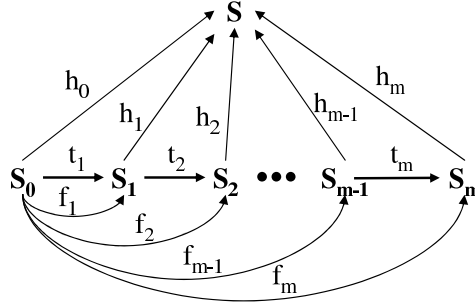


Figure 4.10: Construction and homomorphisms in the proof of decidability.

Second, we define a mapping f_j from S_0 to S_j as a composition $t_j \circ \dots \circ t_0$; the mapping f_j is a homomorphism, because it is a composition of homomorphisms. Initially, $S_0 \models \varphi_\diamond$, according to Lemma 4.5.9. For all S_j , from the existence of a homomorphism f_j from S_0 to S_j we get that $S_j \models \varphi_\diamond$, by Lemma 4.5.7. In particular, $S' \models \varphi_\diamond$.

Third, we show that for all j , h_j defined by the process above is a homomorphism. Initially, $h_0: S_0 \rightarrow S$ is a homomorphism, according to Lemma 4.5.10. If t_j is a merge operation of w_1 and w_2 , then the process applies this operation only if $h_j(w_1) = h_j(w_2)$. From the inductive hypothesis that h_j is a homomorphism, we get that h_{j+1} is a homomorphism, by Lemma 4.5.8.

For every $c[R]p \in \varphi_\square$, if p does not contain positive occurrences of edge formulas or equality formulas, then by Lemma 4.5.7 and the existence of a homomorphism h_r from S' to S , $S' \models c[R]p$, because $S \models c[R]p$.

For the sake of contradiction, assume that the process terminates, but $S' \not\models c[R]p$, where $p(v_0) \stackrel{\text{def}}{=} N(v_0, \dots, v_n) \Rightarrow \psi(v_0, \dots, v_n)$. That is, we can assign nodes w_0, \dots, w_n to v_0, \dots, v_n , respectively, such that there is an R -path from c to w_0 , $N(w_0, \dots, w_n)$ holds but $\psi(w_0, \dots, w_n)$ does not hold. Consider the assignment $h_r(w_0), \dots, h_r(w_n)$ in S . Because homomorphism preserves existences of paths and edges, there is an R -path from c to $h_r(w_0)$, and $N(h_r(w_0), \dots, h_r(w_n))$ holds. Because $S \models c[R]p$, we know that $\psi(w_0, \dots, w_n)$ holds. Therefore, there is an atomic formula θ that appears positively in ψ and evaluates to *false* in S' and to *true* in S .

If θ is an equality formula $v_1 = v_2$, then the merge operation of w_1 and w_2 in S' is enabled (because θ is *false* in S'), and $h(w_1) = h(w_2)$ in S (because θ is *true* in S), contradiction to the assumption that the process terminated. Similarly, if θ is an edge formula $v_1 \xrightarrow{f} v_2$, then the edge-addition operation of w_1 and w_2 in S' is enabled (because θ is *false* in S'), and there is an f -edge from $h(w_1)$ to $h(w_2)$ in S (because θ is *true* in S), contradiction to the assumption that the process terminated. Thus, $S' \models \varphi_\square$.

4.6 The \mathcal{L}_2 Fragment and its Decidability

In this section, we define another fragment of \mathcal{L}_0 , called \mathcal{L}_2 , and show its decidability.

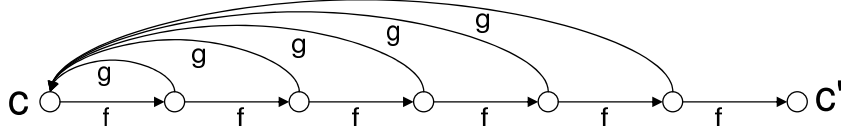
The fragment \mathcal{L}_2 extends \mathcal{L}_1 (defined in Section 4.3) by allowing constants to be freely used in patterns, removing the last restriction of Definition 4.3.1. For example, the property that a general graph is a tree in which each node has a pointer b back to the root is expressible in \mathcal{L}_2 , using the pattern $true \Rightarrow b(v_0, \text{root})$, but this pattern is not in \mathcal{L}_1 . It can be shown that the property cannot be expressed in \mathcal{L}_1 .

In the rest of this section, we explain how to modify the proof of decidability of \mathcal{L}_1 , to prove the decidability of \mathcal{L}_2 . We start by explaining why the proof of Theorem 4.4.14 does not go through for \mathcal{L}_2 . Recall that if a graph is in \mathcal{A}_k , and an operation that is enabled by an \mathcal{L}_1 reachability constraint is applied, then the result is in \mathcal{A}_k , due to the distance restrictions in \mathcal{L}_1 patterns (see Lemma 4.5.5). In \mathcal{L}_2 , this nice property no longer holds.

For example, consider the \mathcal{L}_2 constraint

$$\text{let } p(v_0) \stackrel{\text{def}}{=} (v_0 \xrightarrow{f} v_1) \Rightarrow (v_1 \xrightarrow{g} c) \text{ in } c[\xrightarrow{f}^*]p$$

Given k , we construct a graph G_k that consists of an f -path of $k + 3$ disjoint nodes, but only $k + 1$ nodes on the path have a g -edge back to c . Fig. 4.11 shows G_4 . The graph G_k is in \mathcal{A}_k , but violates the reachability constraint

Figure 4.11: The graph G_4 .

above. Thus, it has an edge-addition operation enabled for adding a g -edge between the first and the last nodes. It is easy to see that after adding the edge, we get a graph G'_k that is not in \mathcal{A}_k .⁹

If the construction of Theorem 4.4.14 is applied to an \mathcal{L}_2 formula, it might generate a graph in which the number of extra edges is proportional to the number of nodes, due to the use of constants in patterns, and not bounded by the size of the formula. The good news is that the extra edges have one of the endpoints labeled with a constant, except, possibly a small number of them. The proof of decidability of \mathcal{L}_2 is based on the fact that each extra edge has one of its endpoints labeled with a constant.

We define a graph operation rem that removes all edges to and from nodes labeled with constants. Formally, the result of $rem(S)$ is a graph S' with the same set of nodes as S , such that there is an f -edge from v_1 to v_2 in S' if and only if there is an f -edge from v_1 to v_2 in S and the nodes v_1 and v_2 are not labeled by any constants in S . \mathcal{A}_k^{rem} is the set of graphs on which rem yields a graph in \mathcal{A}_k , i.e., $\mathcal{A}_k^{rem} \stackrel{\text{def}}{=} \{S \mid rem(S) \in \mathcal{A}_k\}$.

4.6.1 \mathcal{A}_k^{rem} -Model Property of \mathcal{L}_2

We define graph operations enabled by \mathcal{L}_2 formulas (similarly to Section 4.5.3), and prove that \mathcal{A}_k^{rem} is closed under those operations (similarly to Lemma 4.5.5).

Let $p(v_0) \stackrel{\text{def}}{=} N(v_0, \dots, v_n) \Rightarrow \psi(v_0, \dots, v_n)$ be an \mathcal{L}_2 pattern. Let S be a graph, w_1 be a node in S , and $c_2 \in C$.

We say that *edge-addition between w_1 and c_2 is enabled* (by c[R]p) when (a) $(v_1 \xrightarrow{f} c_2)$ (resp. $(c_2 \xrightarrow{f} v_1)$) appears positively in ψ , (b) we can assign nodes w_0, \dots, w_n to v_0, \dots, v_n , respectively, such that there is an R -path from c to w_0 , $N(w_0, \dots, w_n)$ holds, but $\psi(w_0, \dots, w_n)$ does not hold, and (c) there is **no** f -edge from w_1 to the node labeled with c_2 in S (resp. to w_1 from the node labeled with c_2).

Lemma 4.6.1 *Assume that a graph operation is enabled in a graph S by an \mathcal{L}_2 reachability constraint. If $S \in \mathcal{A}_k^{rem}$ then the result of applying the operation is a graph $S' \in \mathcal{A}_k^{rem}$.*

Proof: For graph operations that do not involve constants, the result follows directly from Lemma 4.5.5.

Assume that $S \in \mathcal{A}_k^{rem}$. Suppose that an edge-addition operation between a node w_1 and c_2 is enabled in a graph S . The graph S' is the result of adding the edge between w_1 and the constant c . In this case, $rem(S)$ and $remS'$ is the same graph. Thus, $S' \in \mathcal{A}_k^{rem}$.

Remark. We can show that \mathcal{A}_k^{rem} is closed under merge operations enabled by a pattern with $v_1 = c$. However, this situation never occurs in the construction used in Theorem 4.4.14, because we do not split nodes that are labeled with constants, when we create a pre-model.

The following theorem shows that \mathcal{L}_2 has \mathcal{A}_k^{rem} -property, i.e., every satisfiable \mathcal{L}_2 formula has a model in \mathcal{A}_k^{rem} . The proof is similar to the proof of Theorem 4.4.14, except the use of Lemma 4.6.1 to show that the result $S' \in \mathcal{A}_k^{rem}$.

Theorem 4.6.2 (\mathcal{A}_k^{rem} -Model Property) *If $\varphi \in \mathcal{L}_2$ is satisfiable, then there exists a graph S such that $S \models \varphi$ and $S \in \mathcal{A}_k^{rem}$, where $k = f(\varphi)$ and f is defined in (4.4).*

4.6.2 MSO is decidable on \mathcal{A}_k^{rem}

In this section, we show a reduction from the satisfiability problem of MSO logic on \mathcal{A}_k^{rem} to the satisfiability of MSO on \mathcal{A}_k , which is decidable by Theorem 4.4.8. This reduction completes the proof of decidability of \mathcal{L}_2 .

⁹The tree width of $\mathcal{G}(G_k)$ is k and the tree width of $\mathcal{G}(G'_k)$ is $k + 1$.

Lemma 4.6.3 *There is a translation TR_5 between MSO-formulas such that for every MSO-formula φ , there exists a graph $S \in \mathcal{A}_k^{rem}$ such that $S \models \varphi$ if and only if there exists a graph $S' \in \mathcal{A}_k$ such that $S' \models TR_5(\varphi)$.*

Given the vocabulary $\tau = \langle C, U, F \rangle$ and a number k we define a new vocabulary $\tau' = \langle C, U', F \rangle$, where $U' = U \cup \{F_f^c, B_f^c \mid f \in F, c \in C\}$.

For an MSO formula φ over τ , $TR_5(\varphi)$ is an MSO formula over the vocabulary τ' . The translation TR_5 is defined inductively on φ , as usual. For a binary relation formula $f \in F$, we define:

$$TR_5(f(v_1, v_2)) = (E(v_1, v_2) \wedge F_f(v_2)) \vee (E(v_2, v_1) \wedge B_f(v_1)) \\ \bigvee_{c \in C \cup \{d^1, \dots, d^k\}} (c = v_1 \wedge F_f^c(v_2)) \vee (c = v_2 \wedge B_f^c(v_1))$$

Intuitively, a tree node v is labeled with F_f^c if and only if there is an f -edge from v to the node labeled by c in the corresponding Ayah graph. A tree node v is labeled with B_f^c if and only if there is an f -edge to v from the node labeled by c in the corresponding Ayah graph. This allows us to encode both the direction and the label of the extra edges.

Remark. We have chosen a simple encoding that is not parsimonious in the number of additional unary relations. For example, if an edge has two constants on its adjacent nodes, it can be encoded in more than one way. This ambiguity can be resolved using ordering between constants, but we ignore it here, to simplify the presentation.

Theorem 4.6.4 *The satisfiability problem of MSO formulas is decidable on \mathcal{A}_k^{rem} .*

Proof: Follows from Lemma 4.6.3 and Theorem 4.4.8.

Theorem 4.6.5 *The satisfiability problem of \mathcal{L}_2 is decidable.*

Proof: Follows from combining Theorem 4.4.12, Theorem 4.6.2, Lemma 4.4.2, and Theorem 4.6.4.

4.7 Complexity

In this section, we start with a short discussion of the practical issues related to checking satisfiability of \mathcal{L}_1 formulas. Then, we provide proofs of upper and lower bounds on the worst-case complexity of satisfiability problem for \mathcal{L}_1 .

In Section 4.4, we proved decidability by reduction to MSO on trees, which allows us to check satisfiability of \mathcal{L}_1 formulas using MONA decision procedure [HJJ⁺95]. Alternatively, we can directly construct a tree automaton from an \mathcal{L}_1 formula, and can then check emptiness of the automaton, which yields a double-exponential procedure (Section 4.7.2).

However, a naïve translation of \mathcal{L}_1 formulas to automata does not yield a practical decision procedure. First, the size of the automaton is exponential in the input vocabulary, regardless of the complexity of the input formula. Second, a naïve translation produces *two-way alternating* tree automata. To the best of our knowledge, there are no tools that can check emptiness of such automata. A translation from two-way alternating tree automata to tree automata that can be handled by existing tools, such as MONA [HJJ⁺95], Timbuk [GT01], or H1 [NNS02], is at least exponential.

We are investigating tableaux-based techniques to implement a decision procedure for validity, satisfiability, and model generation for \mathcal{L}_1 . A tableaux-based decision procedure can be adaptive to specific formulas, and the formulas that come up in practice are quite simple.

The lower bound on the complexity of the satisfiability problem of \mathcal{L}_1 is NEXPTIME (Section 4.7.1), but it remains elementary (in contrast to MSO on trees, which is non-elementary [Mey75]). The complexity depends on the bound k of \mathcal{A}_k models, according to Theorem 4.4.14. Finding tighter upper and lower bounds for \mathcal{L}_1 is an open problem.

Bounded-Model Property of \mathcal{L}_1 We can show that \mathcal{L}_1 has a bounded model property: every satisfiable \mathcal{L}_1 formula has a model whose size is a (elementary) function of the size of the formula. The translation of \mathcal{L}_1 formulas to automata and the finite-model property (Theorem 4.1.6) yield a double-exponential bound on the size of a model. We believe that it can be improved. Bounded-model property is important for example for guaranteeing termination of tableaux-based decision procedures.

Bounded Branching of \mathcal{L}_1 Lemma 4.5.11 implies that an upper bound on the branching of a node in a Σ -labeled tree is $r = 2 \times n \times \varphi_\diamond \times |C|$. If a node is not labeled with a constant, we can improve the bound to be $2 \times n \times \varphi_\diamond$. The branching does not increase as a result of merging and edge additions enabled by \mathcal{L}_1 patterns.

Thus, for checking satisfiability of \mathcal{L}_1 it is sufficient to consider only Σ -labeled trees with a branching bounded by r .

The Use of Constants in Routing Expressions If the routing expressions do not contain positive occurrences of constant symbols, then the bound k for \mathcal{L}_1 does not depend on the routing expressions:

Theorem 4.7.1 *Assume that $\varphi \in \mathcal{L}_1$ is satisfiable, and that the routing expressions that appear in φ do not contain positive occurrences of constant symbols. Then, there exists a graph $S \in \mathcal{A}_k$ where $k = |\varphi_\diamond|$, and $S \models \varphi$.*

Sketch of Proof: To prove this, we modify the proof of Theorem 4.4.14. The main observation is that we cannot force a path to visit a node labeled with a constant, except at the endpoints of a path. (a) when creating a pre-model, duplicate nodes with constants, (b) witness splitting results in a pre-model with at most $|\varphi_\diamond|$ extra edges, (c) use homomorphism which only preserves existence of constants, not their absence, and (d) merge operation enabled by \mathcal{L}_1 preserve homomorphism, because they do not require merging a node with a constant, because a pattern may not contain a positive occurrence of equality between a variable and a constant (unlike \mathcal{L}_2).

Constant symbols can be eliminated from routing expressions, but the complexity of this operation is prohibitive. The \mathcal{L}_1 formulas that come up in practice are well-structured, and we hope to achieve a reasonable performance.

4.7.1 Lower Bound: \mathcal{L}_1 is NEXPTIME-hard

In this section, we show that the worst-case complexity of checking satisfiability of \mathcal{L}_1 formulas is at least NEXPTIME. The proof is by reduction from a tiling problem. This proof is an adapted version of the NEXPTIME-hardness proof from [IRR⁺04a, Theorem 5]. In [IRR⁺04a, Theorem 5], universal quantification over nodes is used in the proof. Since this feature is not available in \mathcal{L}_0 , we use here reachability constraints and patterns.

Let \mathcal{T} be a tiling problem as in Definition 4.2.1, and let n be a natural number. It is an NEXPTIME-complete problem to test on input $(\mathcal{T}, 1^n)$ whether there is a \mathcal{T} -tiling of a square grid of size 2^n by 2^n [Pap94].

Theorem 4.7.2 *The satisfiability of \mathcal{L}_1 formulas is NEXPTIME-hard.*

Proof: Let \mathcal{T} be a tiling problem as in Definition 4.2.1, and let n be a natural number. We define a formula φ_n that exactly expresses a solution to the tiling problem. When φ_n is satisfiable, it has a minimal model of size $2^{\Omega(n)}$.

We use two constants: s , denoting the top left node of the grid, and t , denoting the bottom right node of the grid. The desired model will consist of 2^{2n} tiles:

$$\begin{array}{ccc} s = & [1, 1, t_0] & \cdots & [1, 2^n, t] \\ & [2, 1, t'] & \cdots & [2, 2^n, t''] \\ & \vdots & & \vdots \\ & [2^n, 1, t'''] & \cdots & [2^n, 2^n, t_k] = t \end{array}$$

The binary relation n holds between each pair of consecutive tiles, including, for example, $[1, 2^n, t]$ and $[2, 1, t']$. We include the following unary relation symbols: H_1, \dots, H_n , indicating the horizontal position as an n -bit number; V_1, \dots, V_n , indicating the vertical position; and T_0, \dots, T_k , indicating the tile type.

The formula φ_n is the conjunction of the following assertions.

There is a path from s to t :

$$s \langle \xrightarrow{n^*} \rangle t \tag{4.6}$$

All E edges reachable from s are deterministic and unshared:

$$s[\xrightarrow{n^*}] \text{det}_n \wedge \text{uns}_n \tag{4.7}$$

The node labeled with s is the first tile, has tile type t_0 , and the node labeled with t is the last tile and has tile type t_k :

$$T_0(s) \wedge \bigwedge_{i=1}^n (\neg H_i(s) \wedge \neg V_i(s)) \wedge T_k(t) \wedge \bigwedge_{i=1}^n (H_i(t) \wedge V_i(t)) \tag{4.8}$$

We have chosen for simplicity to encode the tile types in unary so we need to say that tile types are mutually exclusive and every node has a tile:

$$s[\underline{n}, *] \left(\bigwedge_{0 \leq i < j \leq k} \neg(T_i \wedge T_j) \right) \wedge \left(\bigvee_{0 \leq i \leq k} T_i \right) \quad (4.9)$$

The arrangement of tiles honors \mathcal{T} 's horizontal and vertical adjacency requirements:

$$\mathbf{let} \ p(v) \stackrel{\text{def}}{=} \text{Next}_h(v, v') \Rightarrow \text{Hor}(v, v') \ \mathbf{in} \ s[\underline{n}, *]p \quad (4.10)$$

$$\mathbf{let} \ p(v) \stackrel{\text{def}}{=} \text{Next}_v(v, v') \Rightarrow \text{Vert}(v, v') \ \mathbf{in} \ s[\underline{n}, *]p \quad (4.11)$$

The abbreviation Next_v , Next_h , Vert , Horz , and Next denote formulas which contain only unary relation symbols and variables, and no equality. We rely on the fact that a neighborhood of a pattern need not be connected.

The abbreviation $\text{Next}_h(x, y)$ means that x and y have the same vertical position and y 's horizontal position is one more than that of x . $\text{Next}_v(x, y)$ means that x and y have the same horizontal position and y 's vertical position is one more than that of x .

$$\begin{aligned} \text{Next}_h(x, y) &\equiv (\bigwedge_{i=1}^n V_i(x) \leftrightarrow V_i(y)) \wedge \text{PlusOne}_h(x, y) \\ \text{Next}_v(x, y) &\equiv (\bigwedge_{i=1}^n H_i(x) \leftrightarrow H_i(y)) \wedge \text{PlusOne}_v(x, y) \end{aligned}$$

The abbreviations $\text{PlusOne}_h(x, y)$ and $\text{PlusOne}_v(x, y)$ are nearly identical. Thus, we restrict our attention to $\text{PlusOne}_h(x, y)$, which means that the horizontal position of y is one greater than the horizontal position of x . (Our convention is that the bit positions are numbered 1 to n , with 1 being the high-order bit, and n the low-order bit.) $\text{PlusOne}_h(x, y)$ can be written as follows:

$$\text{PlusOne}_h(x, y) \equiv \bigvee_{i=1}^n [\bigwedge_{j>i} (H_j(x) \wedge \neg H_j(y)) \wedge (\neg H_i(x) \wedge H_i(y)) \wedge \bigwedge_{j<i} (H_j(x) \leftrightarrow H_j(y))]$$

The length of the formula $\text{PlusOne}_h(x, y)$ is $O(n^2)$.

The abbreviation $\text{Hor}(x, y)$ (resp. $\text{Vert}(x, y)$) is a disjunction over the tile types asserting that the tiles in positions x and y are horizontally (resp. vertically), compatible. For example,

$$\text{Hor}(x, y) \equiv \bigvee_{R(t_i, t_j)} (T_i(x) \wedge T_j(y)) \quad (4.12)$$

The abbreviation $\text{Next}(x, y)$ means $\text{Next}_h(x, y)$ or x has horizontal position 2^n , y has horizontal position 1, and y 's vertical position is one more than that of x :

$$\text{Next}(x, y) \equiv \text{Next}_h(x, y) \vee \left((\bigwedge_{i=1}^n H_i(x)) \wedge (\bigwedge_{i=1}^{n-1} \neg H_i(y)) \wedge H_n(y) \wedge \text{PlusOne}_v(x, y) \right)$$

Finally, if there is an edge from x to y , then there $\text{Next}(x, y)$ holds:

$$\mathbf{let} \ p(v) \stackrel{\text{def}}{=} (v \xrightarrow{n} v' \Rightarrow \text{Next}(v, v')) \ \mathbf{in} \ s[\underline{n}, *]p \quad (4.13)$$

Remark. The length of the formula φ_n described above is $O(n^2)$. The only difficulty in keeping φ_n to total size $O(n)$ is in writing the formulas $\text{PlusOne}_h(x, y)$ and $\text{PlusOne}_v(x, y)$. We can decrease the size by keeping track of the position i using $2n$ addition unary relation symbols, similarly to the proof of [IRR⁺04a, Lemma 14].

4.7.2 Upper Bound: \mathcal{L}_1 is in 2EXPTIME

In this section, we show that the worst case complexity of checking satisfiability of \mathcal{L}_1 formulas is at most double-exponential in the size of the formula. The proof is by reduction to non-emptiness of tree automata. The technique used in this proof is based on ideas for proving an upper bound on the satisfiability problem of two-way μ -calculus [Var98].

A Σ -labeled tree is a pair $\langle T, V \rangle$ where T is a tree (i.e., a connected directed acyclic graph), and V is a mapping that assigns for each node of T a label in Σ .

Definition 4.7.3 (TATA) *Two-way alternating tree automaton (TATA) on finite Σ -labeled trees with branching bounded by r is $\mathcal{A} = \langle \Sigma, Q, \delta, q_0 \rangle$ where Σ is the input alphabet, Q is a finite set of states, $q_0 \in Q$ is an initial state, and $\delta: Q \times \Sigma \rightarrow \mathcal{B}^+(\{-1, 0, \dots, r\} \times Q)$ is the transition function. Here, $\mathcal{B}^+(X)$ is the set of all positive propositional formulas over the propositional variables in the set X .*

A run of \mathcal{A} on a labeled tree $\langle T, V \rangle$ is a labeled tree $\langle T_\rho, \rho \rangle$ in which every node is labeled by an element of $T \times Q$. Intuitively, a node labeled by (x, q) describes a copy of the automaton that is in state q and reads the node x of T . Formally, $\langle T_\rho, \rho \rangle$ satisfies:

- If y is the root of T_ρ then $\rho(y) = (x, q_0)$ where x is the root of T .
- For every $y \in T_\rho$, if $\rho(y) = (x, q)$, and $\delta(q, V(x)) = \theta$, then there is a (possibly empty) set $S = \{(c_1, q_1), \dots, (c_n, q_n)\} \subseteq \{-1, 0, \dots, r\} \times Q$, such that $S \models \theta$, and for all $i = 1, \dots, n$, the node $y \cdot i$ is the i -th successor of the node y in T_ρ , and $\rho(y \cdot i) = (x \cdot c_i, q_i)$. Here, $x \cdot c_i$ denote the c_i -th successor of x in T (when $i > 0$), the node x itself (when $c_i = 0$), or the predecessor of x (when $c_i = -1$ and x is not the root of T).

An automaton accepts a tree t if there exists a (finite) run on t . We denote by $\mathcal{L}(\mathcal{A})$ the set of all Σ -labeled trees that \mathcal{A} accepts.

We start by showing an upper bound for checking satisfiability of normal-form formulas (Section 4.4.3).

Lemma 4.7.4 *For every formula $\varphi \in \mathcal{CL}_1$, there exists a TATA \mathcal{A}_φ , such that $\mathcal{L}(\mathcal{A}_\varphi) = \emptyset$ if and only if φ is unsatisfiable.*

Sketch of Proof: Given $\varphi \in \mathcal{CL}_1$ over the vocabulary $\tau = \langle C, U, F \rangle$, we construct a TATA \mathcal{A}_φ over Σ -labeled trees, defined in Section 4.4.2. Recall from Section 4.4.2 that $\Sigma = \mathcal{P}(C' \cup U')$ where $C' = C \cup \{c^1, \dots, c^k\} \cup \{d^1, \dots, d^k\}$, $U' = \{F_f, B_f, L_f, F_f^{d^i}, B_f^{d^i} \mid f \in F, i = 1, \dots, k\}$, and k is the bound computed from φ using (4.4).

The automaton \mathcal{A}_φ is defined as the intersection of two automata described below. The first automaton \mathcal{A}_1 checks that the labeling of the input tree is legal: every $c \in C'$ appears exactly once in the tree, and that the root of the tree is not labeled by F_f, B_f , for any $f \in F$. The second automaton \mathcal{A}_2 checks that φ is satisfied by the input tree.

We define $\mathcal{A}_1 = \langle \Sigma, Q_1, \delta_1, q_0 \rangle$ where $Q_1 = \{q_0\} \cup \{q_c, q_{-c} \mid c \in C'\}$. In the initial state q_0 , the automaton first checks that the root of the input tree is not labeled by any of F_f, B_f , then it threads into checking each of the constants in C' . Intuitively, when the automaton is in state q_c on node x of the input tree, the subtree rooted at x must contain exactly one node labeled with c . When the automaton is in state q_{-c} on node x , the subtree rooted at x must not contain any node labeled with c . The transition function δ is defined as follows:

$$\begin{aligned} \delta_1(q_0, \sigma) &= \bigwedge_{c \in C'} (0, q_c) && \text{if } \sigma \cap \{F_f, B_f \mid f \in F\} = \emptyset \\ \delta_1(q_c, \sigma) &= \bigvee_{i=1, \dots, r} (i, q_c) \wedge \bigwedge_{j=1, \dots, r, j \neq i} (j, q_{-c}) && \text{if } c \notin \sigma \\ \delta_1(q_c, \sigma) &= \bigwedge_{j=1, \dots, r} (j, q_{-c}) && \text{if } c \in \sigma \\ \delta_1(q_{-c}, \sigma) &= \bigwedge_{j=1, \dots, r} (j, q_{-c}) && \text{if } c \notin \sigma \end{aligned}$$

Before defining \mathcal{A}_2 , we need to introduce some notations. For $\varphi \in \mathcal{CL}_1$, $cl(\varphi)$ denotes the set of all subformulas of the form $c\langle R \rangle c'$ and $c[R]p$ that appear in φ . For a formula $\psi \in cl(\varphi)$, let $R(\psi)$ denote the routing expression that appears in ψ . We use $Patterns(\varphi)$ to denote the set of patterns that occur in φ .

Routing expressions that appear in \mathcal{L}_0 formulas over τ accept finite words over the input vocabulary $\tau^R = \{a, \neg a \mid a \in C \cup U\} \cup \{\vec{f}, \overset{\leftarrow}{f} \mid f \in F\}$. For every routing expression R , there is a nondeterministic finite

automaton on finite words that accepts the same language. For a routing expression R , an automaton A_R is a tuple $\langle \tau^R, Q^R, q_0^R, \delta^R, Q_f^R \rangle$, where Q^R is a finite set of states, $q_0^R \in Q^R$ is the initial state, $Q_f^R \subseteq Q$ is the set of final states, and $\delta^R: Q \times \tau^R \times Q$ is the transition relation. The construction is analogous to the construction of an automaton for regular expressions. The size of the automata is linear in R .

First, we define \mathcal{A}_2 for the case that $k = 0$. The idea is that we can “program” the automaton \mathcal{A}_2 to search for a path in the input tree that satisfies a certain routing expression, using the automaton for that routing expression.

Formally, $\mathcal{A}_2 = \langle \Sigma, Q, \delta, q_0 \rangle$, where

$$Q \stackrel{\text{def}}{=} \{q_0\} \cup \text{cl}(\varphi) \cup \text{Patterns}(\varphi) \cup \{B_f, F_f, \neg F_f, \neg B_f \mid f \in F\} \\ \cup \{(\psi, q) \mid \psi \in \text{cl}(\varphi), q \in Q(A_{R(\psi)})\}$$

The transition function δ is defined as follows.

$$\begin{aligned} \delta(q_0, \sigma) &= \bigwedge_{\psi \in \text{cl}(\varphi)} (0, \psi) \\ \delta(x\langle R \rangle y, \sigma) &= \bigvee_{j=1, \dots, r} (j, x\langle R \rangle y) \quad \text{if } x \notin \sigma \\ \delta(x\langle R \rangle y, \sigma) &= (0, (x\langle R \rangle y, q_0^R)) \quad \text{if } x \in \sigma \\ \delta(x[R]p, \sigma) &= \bigvee_{j=1, \dots, r} (j, x[R]p) \quad \text{if } x \notin \sigma \\ \delta(x[R]p, \sigma) &= (0, (x[R]p, q_0^R)) \quad \text{if } x \in \sigma \end{aligned}$$

For $f \in F$, $\delta(F_f, \sigma) = \text{true}$ if $F_f \in \sigma$ and $\delta(\neg F_f, \sigma) = \text{true}$ if $F_f \notin \sigma$. Similarly, for B_f . For every pattern p that occurs in φ , we can define $\delta(p, \sigma)$ by enumerating all possible neighborhoods of the node. This is exponential in the size of pattern p , but the patterns are usually quite small.

If $y \in \sigma$ and $q \in Q_f^R$, $\delta((x\langle R \rangle y, q), \sigma) = \text{true}$, otherwise

$$\begin{aligned} \delta((x\langle R \rangle y, q), \sigma) = & \\ \bigvee_{\substack{a \in \sigma \\ a \in C \cup U \\ \delta^R(q, a) = q'}} (0, (x\langle R \rangle y, q')) & \quad \bigvee \bigvee_{\substack{a \notin \sigma \\ a \in C \cup U \\ \delta^R(q, -a) = q'}} (0, (x\langle R \rangle y, q')) \\ \bigvee \bigvee_{f \in F} \bigvee_{\delta^R(q, \underline{f}) = q'} (0, B_f) \wedge (-1, (x\langle R \rangle y, q')) & \quad \bigvee \bigvee_{j=1, \dots, r} (j, F_f) \wedge (j, (x\langle R \rangle y, q')) \\ \bigvee \bigvee_{f \in F} \bigvee_{\delta^R(q, \underline{f}) = q'} (0, F_f) \wedge (-1, (x\langle R \rangle y, q')) & \quad \bigvee \bigvee_{j=1, \dots, r} (j, B_f) \wedge (j, (x\langle R \rangle y, q')) \end{aligned}$$

If $q \in Q_f^R$, then $\delta((x[R]p, q), \sigma) = (0, p) \wedge \Delta$, otherwise $\delta((x[R]p, q), \sigma) = \Delta$, where

$$\begin{aligned} \Delta = \bigwedge_{\substack{a \in \sigma \\ a \in C \cup U \\ \delta^R(q, a) = q'}} (0, (x[R]p, q')) & \quad \wedge \bigwedge_{\substack{a \notin \sigma \\ a \in C \cup U \\ \delta^R(q, -a) = q'}} (0, (x[R]p, q')) \\ \wedge \bigwedge_{f \in F, \delta^R(q, \underline{f}) = q'} (0, \neg B_f) \vee (-1, (x[R]p, q')) & \quad \wedge \bigwedge_{j=1, \dots, r} (j, \neg F_f) \vee (j, (x[R]p, q')) \\ \wedge \bigwedge_{f \in F, \delta^R(q, \underline{f}) = q'} (0, \neg F_f) \vee (-1, (x[R]p, q')) & \quad \wedge \bigwedge_{j=1, \dots, r} (j, \neg B_f) \vee (j, (x[R]p, q')) \end{aligned}$$

We can extend the above definition for the case of $k > 0$. Intuitively, if a node is marked with c^i and $F_f^{d^i}$, and we want to traverse an f -edge emanating from that node, the target of such an edge can be a child or a parent of the current node (covered by the definition of δ above), or a node marked with d^i . To handle the latter case, the automaton \mathcal{A}_2 transitions into a special state, in which it searches for the node marked with d^i in the tree. When the node marked with d^i is reached, the automaton \mathcal{A}_2 continues with its previous task. Towards this end, we extend the above definition of \mathcal{A}_2 with the states $\{(d^i, (\psi, q)) \mid i = 1, \dots, k, \psi \in \text{cl}(\varphi), q \in Q(A_{R(\psi)})\}$. A state $(d^i, (\psi, q))$ records the task that should be performed when the node d^i is found. Formally, the traversal is performed by the following transitions:

$$\begin{aligned} \delta((d^i, (\psi, q)), \sigma) &= (0, (\psi, q)) && \text{if } d^i \in \sigma \\ \delta((d^i, (\psi, q)), \sigma) &= (-1, (d^i, (\psi, q))) \vee \bigvee_{j=1, \dots, r} (j, (d^i, (\psi, q))) && \text{otherwise} \end{aligned}$$

Also, δ -transitions of the form $\delta((\psi, q), \sigma)$ are extended to include $(0, (d^i, (\psi, q)))$ as one of the options, when $\delta^R(q, \underline{f}) = q'$ and $F_f^{d^i} \in \sigma$. Similarly, we can handle backwards traversals of edges.

Lemma 4.7.5 *For a fixed τ , and a formula $\varphi \in \mathcal{CL}_1$, let \mathcal{A}_φ be defined as in Lemma 4.7.4. The size of \mathcal{A}_φ is exponential in the size of φ .*

There is a translation from Σ -labeled trees to infinite trees which preserves satisfiability. The emptiness of TATA on infinite trees is DEXPTIME-complete [Var98]. The algorithm for checking emptiness (i) translates from two-way alternating automaton to one-way nondeterministic automaton, which may result in an exponential blowup, and (ii) checks the emptiness of a one-way nondeterministic automaton, which is polynomial in the size of the automaton.

Proposition 4.7.6 *The emptiness problem of TATA on Σ -labeled trees is DEXPTIME-complete.*

This yields a double-exponential upper bound on the complexity of checking satisfiability of normal-form formulas, as stated below.

Theorem 4.7.7 *The satisfiability problem of \mathcal{CL}_1 is in deterministic 2EXPTIME.*

It follows that the upper bound on the (asymptotic) complexity of the satisfiability problem for \mathcal{L}_1 is also double-exponential, as shown below.

Theorem 4.7.8 *The satisfiability problem of \mathcal{L}_1 is in deterministic 2EXPTIME.*

Proof: Given a formula $\varphi \in \mathcal{L}_1$, we use the translation to normal-form, described in Section 4.4.3, to get an equi-satisfiable formula $\varphi_1 \vee \dots \vee \varphi_m$, where for every $i = 1, \dots, m$, the formula φ_i is in \mathcal{CL}_1 , the size of φ_i is at most polynomial in the size of φ , and m is at most exponential in the size of φ . We can check satisfiability of φ by checking satisfiability of every φ_i separately.

Let $|\varphi|$ denote the size of the formula φ . For every $i = 1, \dots, m$, checking satisfiability of φ_i is $\mathcal{O}(2^{2^{|\varphi|}})$, according to Theorem 4.7.7. Therefore, checking satisfiability of φ is $2^{|\varphi|} \times \mathcal{O}(2^{2^{|\varphi|}})$, which is also $\mathcal{O}(2^{2^{|\varphi|}})$. That is, the complexity of checking satisfiability of \mathcal{L}_1 formulas is in deterministic 2EXPTIME.

4.8 Limitations and Further Extensions

Despite the fact that \mathcal{L}_2 is useful, there are interesting program properties that cannot be expressed directly. For example, transitivity of a binary relation, that can be used, e.g., to express partial orders, is naturally expressible in \mathcal{L}_0 , but not in \mathcal{L}_2 . There are of course interesting properties that are beyond \mathcal{L}_0 , such as the property that a general graph is a tree in which every leaf has a pointer to the root of a tree.

In the future, we plan to generalize \mathcal{L}_2 while maintaining decidability, perhaps beyond \mathcal{L}_0 (i.e., to capture properties that are not expressible in \mathcal{L}_0). We are encouraged by the fact that the proof of decidability in Section 4.4 holds “as is” for many useful extensions. For example, more complex patterns can be used, as long as they do not violate the \mathcal{A}_k -model property.

4.8.1 The Logic \mathcal{L}_3

In the \mathcal{L}_0 logic, reachability constraints describe paths that start from nodes labeled by some constant. The requirement that a path start with a constant is not necessary for decidability. We define \mathcal{L}_3 that generalizes \mathcal{L}_0 with paths that start from any node that satisfies a quantifier-free *positive* formula θ :

$$\theta[R]p \stackrel{\text{def}}{=} \forall w_0, \dots, w_m, v_0, \dots, v_n. R(w_0, v_0) \wedge \theta(w_0, \dots, w_m) \Rightarrow p(v_0, \dots, v_n)$$

A simple and very useful fragment of \mathcal{L}_3 is \mathcal{L}_4 in which θ is fixed to be *true*. We use $[R]p$ to denote *true* $[R]p$. For example, we can specify that all f -edges in the graph are deterministic, and not only those reachable from some constant: $[\epsilon]det_f$.

The fragment \mathcal{L}_3 provides several ways to express the same property; this flexibility can be useful when writing specifications manually. For example, the formula $(x \vee y)[R]p$ in \mathcal{L}_3 is equivalent to $x[R]p \vee y[R]p$ in \mathcal{L}_1 , and to $[x + y.R]p$ in \mathcal{L}_4 . The formula $(x \wedge y)[R]p$ in \mathcal{L}_3 is equivalent to $(x = y) \Rightarrow x[R]p$ in \mathcal{L}_1 and to $[x.y.R]p$ in \mathcal{L}_4 .

We can translate every \mathcal{L}_0 formula to \mathcal{L}_4 using constants in routing expressions: $x[R]p \in \mathcal{L}_0$ is translated into $[x.R]p$. We can show that \mathcal{L}_3 has a finite model property. The logic *LRP* that results from \mathcal{L}_3 by restricting it to \mathcal{L}_2 patterns is decidable.

4.8.2 The Logic UL_1

We can extend \mathcal{L}_1 with (a possibly restricted use of) quantifiers, going beyond the proposition logic \mathcal{L}_0 . This extension provides a more general way to write specifications.

We extend \mathcal{L}_1 with universal quantification over constants, as follows. For a vocabulary τ , a formula in UL_1 over τ is a positive boolean combination of formulas of the form $\forall c_1, \dots, c_n. \varphi'$, where φ' is in \mathcal{L}_1 over the vocabulary $\tau' = \tau \cup \{c_1, \dots, c_n\}$. The semantics of the universal quantifiers is defined as usual. The problem of validity of UL_1 -formulas is decidable by reduction to validity in \mathcal{L}_1 .

Lemma 4.8.1 *Let $\varphi \in UL_1$ be of the form $\forall c_1, \dots, c_n. \varphi'$. The formula φ is valid if and only if φ' is valid.*

Note that UL_1 is *not* closed under negation (whereas \mathcal{L}_1 is closed under negation).

It is possible to add quantification over sets and relations, while preserving decidability, as long as there are no quantifier alternations. Quantification of binary relations can be useful for writing modular specifications, and analysis that does not violate abstraction layers. For example, if a procedure's formal parameter x is a pointer to an abstract data-type, we can specify that the field of objects that implement the abstract data-type are not modified by the procedure, without exposing the implementation: $\forall \Sigma. \forall f, f'. x[\Sigma^*] \text{same}_{f, f'}$.

4.9 Related Work

There are several works on logic-based frameworks for reasoning about graph/heap structures. We mention here the ones which are, as far as we know, the closest to ours.

The logic \mathcal{L}_0 can be seen as a fragment of the first-order logic over graph structures with transitive closure (TC logic [Imm87]). It is well known that TC is undecidable, and that this fact holds even when transitive closure is added to simple fragments of FO such as the decidable fragment L^2 of formulas with two variables [Mor75, GKV97, GME99].

It can be seen that our logics \mathcal{L}_0 and \mathcal{L}_1 are both uncomparable with $L^2 + TC$. Indeed, in \mathcal{L}_0 no alternation between universal and existential quantification is allowed. On the other hand, \mathcal{L}_1 allows us to express patterns (e.g., heap sharing) that require more than two variables (see Fig. 4.2, Section 4.3).

In [BRS99], decidable logic L_r (which can also be seen as a fragment of TC) is introduced. The logics \mathcal{L}_0 and \mathcal{L}_1 generalize L_r , which is in fact the fragment of these logics where only two fixed patterns are allowed: equality to a program variable and heap sharing.

In [IRR⁺04a, BPZ05, LQ06, BCO04] other decidable logics are defined, but their expressive power is rather limited w.r.t. \mathcal{L}_1 since they allow at most one binary relation symbol (modelling linked data-structures with 1-selector). For instance, the logic of [IRR⁺04a] does not allow us to express the reversal of a list. Concerning the class of 1-selector linked data-structures, [BI05] provides a decision procedure for a logic with reachability constraints and arithmetical constraints on lengths of segments in the structure. It is not clear how the proposed techniques can be generalized to larger classes of graphs. Other decidable logics [BIL04, KR04] are restricted in the sharing patterns and the reachability they can describe.

Other works in the literature consider extensions of the first-order logic with fixed point operators. Such an extension is again undecidable in general but the introduction of the notion of (loosely) guarded quantification allows one to obtain decidable fragments such as μGF (or μLGF) (Guarded Fragment with least and greater fixed point operators) [GW99, Grä02]. Similarly to our logics, the logic μGF (and also μLGF) has the tree model property: every satisfiable formula has a model of bounded tree width. However, guarded fixed point logics are incomparable with \mathcal{L}_0 and \mathcal{L}_1 . For instance, the \mathcal{L}_1 pattern det_f that requires determinism of f -field, is not a (loosely) guarded formula.

The PALE system [MS01] uses an extension of the weak monadic second order logic on trees as a specification language. The considered linked data-structures are those that can be defined as *graph types* [KS93]. Basically, they are graphs that can be defined as trees augmented by a set of edges defined using routing expressions (regular expressions) defining paths in the (undirected structure of the) tree. \mathcal{L}_1 allows us to reason naturally about arbitrary graphs without limitation to tree-like structures. By restricting the syntax, we guarantee that satisfiability queries posed over arbitrary graphs can be answered precisely by considering only tree-like graphs. This approach allows us to automate the reasoning about limited but interesting properties of *arbitrary* graphs.

Moreover, as we show in Section 4.3, our logical framework allows us to express postconditions and loop invariants that relate the input and the output state. For instance, even in the case of singly-linked lists, our framework allows us to express properties that cannot be expressed in the PALE framework: in the list reversal example of Section 4.3, we show that the output list is precisely the reversed input list, by expressing the relationships between fields before and after the procedure, whereas in the PALE approach, a postcondition can only express that the output is a list that is a permutation of the input list. In particular, a postcondition that relates fields before and after the procedure involves two binary relations with arbitrary interpretation. This can be easily done in \mathcal{L}_0 which supports an arbitrary number of binary relations. This is not supported by PALE, which allows two binary relations with a specific interpretation as tree edges. In the PALE approach, a postcondition can only express that the output is a list that is a permutation of the input list.

In [IRR⁺04b], we tried to employ a decision procedure for MSO on trees to reason about reachability. However, this places a heavy burden on the specifier to prove that the data-structures in the program can be simulated using trees. Our work aims at eliminating this burden by defining syntactic restrictions on the formulas and showing a general reduction theorem.

Other approaches in the literature use undecidable formalisms such as [HHN92], which provides a natural and expressive language, but does not allow for automatic property checking.

Separation logic has been introduced recently as a formalism for reasoning about heap structures [Rey02]. The general logic is undecidable [CYO01] but there are few works showing decidable fragments [CYO01, BCO04]. One of the fragments is propositional separation logic where quantification is forbidden [CYO01, CGH05] and therefore seems to be incomparable with our logic. The fragment defined in [BCO04] allows one to reason only about singly-linked lists with explicit sharing. In fact, the fragment considered in [BCO04] can be translated to \mathcal{L}_1 , and therefore, entailment problems as stated in [BCO04] can be reduced to validity of implications in \mathcal{L}_1 .

The logic \mathcal{L}_0 integrates features of such prominent formalisms as the modal logics, the classical first-order logic, and the regular expressions. The hybrid logics [ABM01] also combine features of modal and classical logics. The most relevant is the hybrid μ -calculus [SV01] which extends the μ -calculus with the following features: (i) nominals, that correspond to constants in \mathcal{L}_1 , (ii) universal program, that corresponds to the fragment \mathcal{L}_4 , and (iii) the ability to reasoning about the past, that corresponds to the use of backward edges in routing expressions. The hybrid μ -calculus is incomparable in its expressive power to \mathcal{L}_1 : on one hand, it supports a more general reachability via the least and greatest fixed point operators; on the other hand, the equality is restricted to nominals. For example, it cannot express that a graph is a tree. Unlike \mathcal{L}_0 , the hybrid μ -calculus does not have a finite model property. Every satisfiable formula in hybrid μ -calculus has a tree-like model. The complexity of hybrid μ -calculus is EXPTIME-complete, but currently, there is no practical decision procedure available. Reportedly, a tableaux-based decision procedure for the alternation-free fragment of hybrid μ -calculus is being developed.

\mathcal{L}_0 shares some common features with description logics [ea03], which is traditionally used for knowledge representation, databases, semantic web, with the notable exception of [GM05], which shows the description logics can be used for reasoning about data-structures. The basic notions of Description Logics are concepts, that correspond to unary relations in \mathcal{L}_0 , and roles, that correspond to binary relations in \mathcal{L}_1 . In addition, expressive Description Logics support (iii) nominals, that correspond to constants in \mathcal{L}_0 ; quantified role restrictions, that can encode determinism; and inverse roles, that correspond to backward edges in routing expressions. The combination of quantified role restrictions and inverse roles provides a way to express sharing. The need for transitivity and fixed points arises in many contexts [CGL99], including, service description logics [Bon02]. It has been shown that a description logic which combines with nominals, inverse roles, determinism, and least fixed points is undecidable [BP04]. In light of the negative results, it is interesting to investigate the usefulness of \mathcal{L}_1 for specifying web services. There are a variety of efficient reasoning tools for description logics, both tableaux-based and resolution-based, which provide some support for expressive features, such as nominals and inverse roles, e.g., FaCT, Racer. To the best of our knowledge, none of the existing tools supports transitive closure of roles or fixed points.

Chapter 5

Conclusions and Future Work

This thesis explores several ways in which program analysis and verification can benefit from employing theorem provers. While these algorithms are applicable to a wide range of analysis problems, the main focus of this thesis is analysis of programs that manipulate linked data-structures.

In Chapter 2, we presented a novel algorithm that computes abstract representation of reachable program states using a novel combination of concrete execution, abstraction, and an automatic theorem prover. Our method complements existing techniques that combine dynamic and static analysis in that it is oriented towards finding a proof rather than finding errors. We leverage existing test suites and fabricated states to speed up the analysis and to reduce the cost of a theorem prover.

This work suggests several interesting directions of research, including the use of fabricated states to (i) generate useful test inputs, (ii) classify potential errors into false alarms and real errors, and (iii) guide abstraction refinement.

In Chapter 3, we presented an algorithm that is specialized for canonical abstraction, and thus, for reasoning about linked data-structures. This algorithm solves several open problems in shape analysis, including the problems of (i) computing the most-precise abstraction of the set of states that are represented by a and satisfy a precondition φ , and (ii) implementing best abstract transformers.

An important issue is the definition of an appropriate *specification language* that is both expressive enough to describe invariants of linked data-structures and amenable to automated reasoning. These invariants often involve reachability between objects in memory and sharing, i.e., aliasing of pointers and object fields deep in the data-structures. Automated reasoning about the combination of these properties is usually undecidable and unpredictable, with *LRP* being one of the rare exceptions.

The decidability result for *LRP*, presented in Chapter 4, improves the state-of-the-art significantly. In contrast to [IRR⁺04a, BPZ05, LQ06, BCO04], *LRP* allows several binary relations. This provides a natural way to (i) specify invariants for data-structures with multiple fields (e.g., trees, doubly-linked lists), (ii) specify post-conditions for procedures that mutate pointer fields of data-structures, by expressing the relationships between fields before and after the procedure (e.g., list reversal, which is beyond the scope of PALE [MS01]), (iii) express verification conditions using a copy of the vocabulary for each program location. Operating on general graphs allows us to verify that the data-structure invariant is reestablished after a sequence of low-level mutations that temporarily violate the data-structure invariant.

Defining decidable fragments of first-order logic with transitive closure over arbitrary graphs is a difficult task (e.g., [IRR⁺04a]). In Chapter 4, we demonstrated that this is possible by combining the following principles:

- Allow arbitrary boolean combinations of the reachability constraints, which are closed formulas without quantifier alternations.
- Define reachability using regular expressions denoting pointer access paths (not) reaching a certain pattern.
- Syntactically limit the way patterns are formed. Extensions of the patterns that allow larger distances between nodes in the pattern either break our proof of decidability or are directly undecidable.

Interestingly, reachability and sharing are important properties in an entirely different context, namely the *semantic web*. For example, both reachability and sharing properties can appear in a description of the functional behavior of e-Services. These properties fall within some very expressive description logics, which are undecidable [BP04]. To the best of our knowledge, there is no decidable description logic which covers both of these

properties, and these properties cannot be handled by existing tools for description logics. It suggests that a decision procedure for *LRP* [YRS⁺06] can be useful in the context of the semantic web. We plan to investigate this relationship further.

Perhaps the most exciting future application of the results described in this thesis is modular analysis. The idea of modular analysis is to exploit the modularity of software systems. Complex software systems are necessarily composed of numerous modules, reusable components, and layers of abstraction. When the module boundaries and interactions between modules are specified by the user (software designer or program developer), each module can be analyzed in isolation using a precise analysis.

In this setting, a user writes specifications that are later used by an automatic program analysis tool to reason about the program. The problem is that user-provided specifications reason about properties of *concrete* program states directly, whereas program analysis operates on *abstract* representation of sets of concrete program states. In other words, there is a gap between specifications written by humans and specifications consumed by program analysis tools: they reason at different levels of abstraction.

This thesis provides a way to bridge the gap by

- (a) assisting program analysis tools in reasoning about human-provided specifications, and
- (b) developing specification languages that are natural for writing specification, and can be incorporated in automatic program analyses.

Bibliography

- [ABM01] C. Areces, P. Blackburn, and M. Marx. Hybrid logics: characterization, interpolation and complexity. *The Journal of Symbolic Logic*, 66(3):977–1010, 2001.
- [ALS91] S. Arnborg, J. Lagergren, and D. Seese. Easy problems for tree-decomposable graphs. *J. Algorithms*, 12(2):308–340, 1991.
- [AMSS06] G. Arnold, R. Manevich, M. Sagiv, and R. Shaham. Combining shape analyses by intersecting abstractions. In *VMCAI*, pages 33–48, 2006.
- [Avr03] A. Avron. Transitive closure and the mechanization of mathematics. In *Thirty Five Years of Automating Mathematics*, pages 149–171. Kluwer Academic Publishers, 2003.
- [Bal04] T. Ball. A theory of predicate-complete test coverage and generation. In *3rd International Symposium on Formal Methods for Components and Objects*, 2004.
- [BCC⁺05] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of jml tools and applications. *Int. J. on Software Tools for Technology Transfer*, 7(3):212–232, 2005.
- [BCO04] J. Berdine, C. Calcagno, and P. O’Hearn. A Decidable Fragment of Separation Logic. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*. LNCS 3328, 2004.
- [BFT05] P. Baumgartner, A. Fuchs, and C. Tinelli. Implementing the Model Evolution Calculus. In Stephan Schulz, Geoff Sutcliffe, and Tanel Tammet, editors, *Special Issue of the International Journal of Artificial Intelligence Tools (IJAIT)*, International Journal of Artificial Intelligence Tools, 2005. Preprint.
- [BHPV05] A. Bouajjani, P. Habermehl, P. Moro, and T. Vojnar. Verifying programs with dynamic 1-selector-linked structures in regular model checking. In *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3440 of LNCS. Springer-Verlag, 2005.
- [BI05] M. Bozga and R. Iosif. Quantitative Verification of Programs with Lists. In *VISSAS intern. workshop*. IOS Press, 2005.
- [BIL04] M. Bozga, R. Iosif, and Y. Lakhnech. On logics of aliasing. In *Static Analysis Symp.*, pages 344–360, 2004.
- [BLM05] T. Ball, S. K. Lahiri, and M. Musuvathi. Zap: Automated theorem proving for software analysis. In *LPAR*, pages 2–22, 2005.
- [BMMR01] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of c programs. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 203–213, 2001.
- [Bon02] P. A. Bonatti. Towards service description logics. In *JELIA*, pages 74–85, London, UK, 2002. Springer-Verlag.
- [BP04] P. A. Bonatti and A. Peron. On the undecidability of logics with converse, nominals, recursion and counting. *Artificial Intelligence*, 158(1):75–96, 2004.

- [BPZ05] I. Balaban, A. Pnueli, and L. D. Zuck. Shape analysis by predicate abstraction. In *VMCAI*, pages 164–180, 2005.
- [BR01] T. Ball and S.K. Rajamani. The SLAM toolkit. In *Int. Conf. on Computer Aided Verification (CAV)*, Lec. Notes in Comp. Sci., pages 260–264, 2001.
- [BRS99] M. Benedikt, T. Reps, and M. Sagiv. A decidable logic for describing linked data structures. In *European Symp. On Programming*, pages 2–19, March 1999.
- [CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 238–252, New York, NY, 1977. ACM Press.
- [CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 269–282, New York, NY, 1979. ACM Press.
- [CGH05] C. Calcagno, P. Gardner, and M. Hague. From Separation Logic to First-Order Logic. In *Foundations of Software Science and Computation Structures (FoSSaCS)*. LNCS 3441, 2005.
- [CGL94] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *Trans. on Prog. Lang. and Syst.*, 16(5):1512–1542, 1994.
- [CGL99] D. Calvanese, G. De Giacomo, and M. Lenzerini. Reasoning in expressive description logics with fixpoints based on automata on infinite trees. In *IJCAI*, pages 84–89, 1999.
- [CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 84–96, 1978.
- [Cou89] B. Courcelle. The monadic second-order logic of graphs, ii: Infinite graphs of bounded width. *Mathematical Systems Theory*, 21(4):187–221, 1989.
- [CS03] K. Claessen and N. Sorensson. New techniques that improve mace-style finite model finding. In *CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications*, 2003.
- [CS04] C. Csallner and Y. Smaragdakis. Jcrasher: an automatic robustness tester for java. *Softw., Pract. Exper.*, 34(11):1025–1050, 2004.
- [CS05] C. Csallner and Y. Smaragdakis. Check 'n' crash: combining static checking and testing. In *ICSE*, pages 422–431, 2005.
- [CYO01] C. Calcagno, H. Yang, and P. O’Hearn. Computability and Complexity Results for a Spatial Assertion Language for Data Structures. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*. LNCS 2245, 2001.
- [Dam96] D. Dams. *Abstract Interpretation and Partial Refinement for Model Checking*. PhD thesis, Technical Univ. of Eindhoven, Eindhoven, The Netherlands, July 1996.
- [Die00] Reinhard Diestel. *Graph Theory*. Springer-Verlag, 2000. Electronic Edition.
- [Dij76] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
- [DNS03] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, 2003.
- [DOY06] D. Distefano, P. O’Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, pages 287–302, 2006.
- [ea03] F. Baader et al., editor. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

- [ECGN01] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2):1–25, February 2001.
- [Ere04] G. Erez. Generating concrete counter examples for arbitrary abstract domains. Master’s thesis, Tel-Aviv University, Israel, 2004.
- [Ern03] M. D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, Portland, OR, May 9, 2003.
- [FLL⁺02] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended static checking for java. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, 2002.
- [Flo67] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics 19*, pages 19–32, Providence, 1967. American Mathematical Society.
- [GHK⁺06] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. Synergy: a new algorithm for property checking. In *SIGSOFT FSE*, pages 117–127, 2006.
- [GKS05] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 213–223, 2005.
- [GKV97] E. Grädel, P. Kolaitis, and M. Vardi. On the decision problem for two variable logic. *Bulletin of Symbolic Logic*, 1997.
- [GM05] L. Georgieva and P. Maier. Description logics for shape analysis. In *SEFM*, pages 321–331, 2005.
- [GME99] E. Grädel, M. Otto, and E. Rosen. Undecidability results on two-variable logics. *Archive of Math. Logic*, 38:313–354, 1999.
- [Grä02] E. Grädel. Guarded fixed point logic and the monadic theory of trees. *Theoretical Computer Science*, 288:129–152, 2002.
- [GS97] S. Graf and H. Säidi. Construction of abstract state graphs with PVS. In *Int. Conf. on Computer Aided Verification (CAV)*, LNCS 1254, pages 72–83. Springer-Verlag, June 1997.
- [GT01] T. Genet and V. Tong. Reachability analysis of term rewriting systems with timbuk. In *LPAR*, pages 695–706, 2001.
- [GTS05] W. Grieskamp, N. Tillmann, and W. Schulte. XRT – exploring runtime for .NET: Architecture and applications. In *SoftMC*, 2005.
- [GW99] E. Grädel and I. Walukiewicz. Guarded Fixed Point Logic. In *Logic in Computer Science (LICS)*. IEEE, 1999.
- [Har00] M. J. Harrold. Testing: a roadmap. In *ICSE - Future of SE Track*, pages 61–72, 2000.
- [Hen90] L. Hendren. *Parallelizing Programs with Recursive Data Structures*. PhD thesis, Cornell Univ., Ithaca, NY, Jan 1990.
- [HHN92] L. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and the transformation of imperative programs. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 249–260, New York, NY, June 1992. ACM Press.
- [HJJ⁺95] J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 1995.
- [HJMS03] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast. In *SPIN*, pages 235–239, 2003.

- [Hoa75] C.A.R. Hoare. Recursive data structures. *Int. J. of Comp. and Inf. Sci.*, 4(2):105–132, 1975.
- [Hol03] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.
- [Imm87] N. Immerman. Languages that capture complexity classes. *SIAM Journal of Computing*, 16:760–778, 1987.
- [IO01] S. S. Ishtiaq and P. W. O’Hearn. Bi as an assertion language for mutable data structures. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 14–26, 2001.
- [IRR⁺04a] N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. The boundary between decidability and undecidability for transitive-closure logics. In *Int. Conf. on Computer Science and Logic (CSL)*, pages 160–174, 2004.
- [IRR⁺04b] N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. Verification via structure simulation. In *Int. Conf. on Computer Aided Verification (CAV)*, pages 281–294, 2004.
- [KR04] V. Kuncak and M. Rinard. Generalized records and spatial conjunction in role logic. In *Static Analysis Symp.*, Verona, Italy, August 26–28 2004.
- [KS93] N. Klarlund and M. Schwartzbach. Graph types. In *ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press, 1993.
- [LAIR⁺05] T. Lev-Ami, N. Immerman, T. Reps, M. Sagiv, S. Srivastava, and G. Yorsh. Simulating reachability using first-order logic with applications to verification of linked data structures. In *Conference on Automated Deduction (CADE)*, pages 99–115, 2005.
- [LAIS06] T. Lev-Ami, N. Immerman, and M. Sagiv. Abstraction for shape analysis with fast and precise transformers. In *CAV*, pages 547–561, 2006.
- [LARSW00] T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *ISSTA*, pages 26–38, 2000.
- [LAS00] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symp.*, pages 280–301, 2000. The system is available from www.cs.tau.ac.il/~tvla.
- [LNS00] K. R. M. Leino, G. Nelson, and J. B. Saxe. Esc/java users manual. Technical Report 002, Compaq Systems Research Center, 2000.
- [Log04] F. Logozzo. *Modular Static Analysis of Object Oriented Languages*. PhD thesis, LEcole Polytechnique, 2004.
- [LQ06] S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2006.
- [LY92] D. Lee and M. Yannakakis. Online minimization of transition systems (extended abstract). In *STOC*, pages 264–274, 1992.
- [LYY05] O. Lee, H. Yang, and K. Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *ESOP*, pages 124–140, 2005.
- [MBC⁺07] R. Manevich, J. Berdine, B. Cook, G. Ramalingam, and M. Sagiv. Shape analysis by graph decomposition. In *TACAS*, pages 3–18, 2007.
- [Mey75] Albert R. Meyer. Weak monadic second-order theory of successor is not elementary recursive. In *Logic Colloquium (Proc. Symposium on Logic, Boston, 1972)*, volume 453, pages 132–154, 1975.
- [MLK98] J. S. Moore, T. W. Lynch, and M. Kaufmann. A mechanically checked proof of the amd5_k86tm floating point division program. *IEEE Trans. Computers*, 47(9):913–926, 1998.
- [Mor75] M. Mortimer. On languages with two variables. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 21:135–140, 1975.

- [MPC⁺02] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. Cmc: A pragmatic approach to model checking real code. In *OSDI*, 2002.
- [MS01] A. Møller and M.I. Schwartzbach. The pointer assertion logic engine. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 221–231, 2001.
- [MYRS05] R. Manevich, E. Yahav, G. Ramalingam, and S. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In *VMCAI*, pages 181–198, 2005.
- [NE02] J. W. Nimmer and M. D. Ernst. Invariant inference for static checking: An empirical evaluation. In *FSE 2002*, pages 11–20, 2002.
- [NNH99] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
- [NNS02] F. Nielson, H. R. Nielson, and H. Seidl. Normalizable horn clauses, strongly recognizable relations, and spi. In *SAS*, pages 20–35, 2002.
- [Pap94] C. M. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994.
- [PE05] Carlos Pacheco and Michael D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP*, pages 504–527, 2005.
- [PPV05] C. Pasareanu, R. Pelanek, and W. Visser. Concrete model checking with abstract matching and refinement. In *CAV*, 2005.
- [Rab69] M. Rabin. Decidability of second-order theories and automata on infinite trees. *Trans. Amer. Math. Soc.*, 141:1–35, 1969.
- [Rey02] J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Logic in Computer Science (LICS)*. IEEE, 2002.
- [RS86] N. Robertson and P. D. Seymour. Graph minors. ii. algorithmic aspects of tree-width. *J. Algorithms*, 7(3):309–322, 1986.
- [RSW04] T. Reps, M. Sagiv, and R. Wilhelm. Static program analysis via 3-valued logic. In *CAV*, pages 15–30, 2004.
- [RSY04] T. Reps, M. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *Int. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 252–266, 2004.
- [RV01] A. Riazanov and A. Voronkov. Vampire 1.1 (system description). In *IJCAR*, pages 376–380, 2001.
- [See92] D. Seese. Interpretability and tree automata: A simple way to solve algorithmic problems on graphs closely related to trees. In *Tree Automata and Languages*, pages 83–114. North-Holland, 1992.
- [SLA02] D. Stotts, M. Lindsey, and A. Antley. An informal formal method for systematic junit test case generation. In *XP/Agile Universe*, pages 131–143, 2002.
- [SMA05] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *ESEC/SIGSOFT FSE*, pages 263–272, 2005.
- [SP81] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.
- [SRW98] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *Trans. on Prog. Lang. and Syst.*, 20(1):1–50, January 1998.
- [SRW99] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 105–118, New York, NY, January 1999. ACM Press.

- [SRW02] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *Trans. on Prog. Lang. and Syst.*, 2002.
- [SV01] U. Sattler and M. Y. Vardi. The hybrid μ -calculus. In *IJCAR*, pages 76–91, 2001.
- [Var98] M. Y. Vardi. Reasoning about the past with two-way automata. In *ICALP*, pages 628–641, 1998.
- [VHB⁺03] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.
- [Wei] C. Weidenbach. SPASS: An automated theorem prover for first-order logic with equality. Available at “<http://spass.mpi-sb.mpg.de/index.html>”.
- [XN03] T. Xie and D. Notkin. Tool-assisted unit test selection based on operational violations. In *ASE*, pages 40–48, 2003.
- [YBS06] G. Yorsh, T. Ball, and M. Sagiv. Testing, abstraction, theorem proving: better together! In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 145–156, 2006.
- [YBS07] G. Yorsh, T. Ball, and M. Sagiv. Testing, abstraction, theorem proving: better together! *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2007. Invited, submitted.
- [Yor03] G. Yorsh. Logical characterizations of heap abstractions. Master’s thesis, Tel-Aviv University, Tel-Aviv, Israel, 2003. Available at “<http://www.cs.tau.ac.il/~gretay>”.
- [YRS04] G. Yorsh, T. Reps, and M. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 530–545, 2004.
- [YRS⁺06] G. Yorsh, A. Rabinovich, M. Sagiv, A. Meyer, and A. Bouajjani. A logic of reachable patterns in linked data-structures. In *Foundations of Software Science and Computation Structures (FoSSaCS)*, pages 94–110, 2006.
- [YRS⁺07] G. Yorsh, A. Rabinovich, M. Sagiv, A. Meyer, and A. Bouajjani. A logic of reachable patterns in linked data-structures. *J. Log. Algebr. Program. (JLAP)*, 73(1-2):111–142, 2007.
- [YRSW07] G. Yorsh, T. Reps, M. Sagiv, and R. Wilhelm. Logical characterizations of heap abstractions. *ACM Transactions on Computational Logic (TOCL)*, 8, January 2007.
- [ZHM97] H. Zhu, P.A. Hall, and H.R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):336–427, December 1997.

Appendix A

Appendix for Chapter 2

A.1 Lattice operations

Let \mathcal{A} be a set with partial order \sqsubseteq . An element $a \in \mathcal{A}$ is a **lower bound** of a set $X \subseteq \mathcal{A}$ if, for a $x \in X$, $a \sqsubseteq x$. The **meet operator**, denoted by \sqcap , yields the greatest lower bound with respect to \sqsubseteq ; i.e., for a set $X \subseteq \mathcal{A}$, $\sqcap X$ is a lower bound of X , and for every lower bound a of X , $a \sqsubseteq \sqcap X$. Similarly, an element $a \in \mathcal{A}$ is an **upper bound** of a set $X \subseteq \mathcal{A}$ if, for every $x \in X$, $x \sqsubseteq a$. Similarly, the **join operator**, denoted by \sqcup , yields the least upper bound with respect to \sqsubseteq ; i.e., for every set $X \subseteq \mathcal{A}$, $\sqcup X$ is an upper bound of X , and for every upper bound a of X , $\sqcup X \sqsubseteq a$.

A complete lattice is a partially-ordered set in which every subset has both least upper bound and greatest lower bound.

A widening operator on \mathcal{A} is defined as a (partial) function $\nabla: \mathcal{A} \times \mathcal{A} \rightarrow \mathcal{A}$ satisfying: (i) for each $x, y \in \mathcal{A}$, $x \sqsubseteq x \nabla y$ and $y \sqsubseteq x \nabla y$; and (ii) for all increasing chains $y_0 \sqsubseteq y_1 \sqsubseteq \dots$ the increasing chain defined by $x_0 \stackrel{\text{def}}{=} y_0$ and $x_{i+1} \stackrel{\text{def}}{=} x_i \nabla y_{i+1}$ is not strictly increasing.

A.2 Proofs

In this section, we provide (straightforward) proofs of the theorems stated in Chapter 2.

Let D be a complete lattice with \sqsubseteq, \sqcap , and \sqcup operations. A function $h: D \rightarrow D$ is monotone if and only if for all d, d' , if $d \sqsubseteq d'$ then $h(d) \sqsubseteq h(d')$. A function $h: D \rightarrow D$ is extensive if and only if for all d , $d \sqsubseteq h(d)$. We use $\text{LFP}(h)$ to denote the least fixed point of h , i.e., the element $d \in D$ such that $h(d) = d$ and for all $d' \in D$, if $h(d') = d'$ then $d \sqsubseteq d'$. Also, for $i \in D$, $\text{LFP}_{\sqsupseteq i}(h)$ denotes the least fixed point of h w.r.t. i , i.e., the element $d \in D$ such that $h(d) = d$, $i \sqsubseteq d$, and for all $d' \in D$, if $h(d') = d'$ and $i \sqsubseteq d'$, then $d \sqsubseteq d'$.

Theorem A.2.1 (Tarski Theorem, 1955) *Let D be a complete lattice and $f: D \rightarrow D$ be a monotone function.*

$$\text{LFP}(f) = \sqcap \text{Fix}(f) = \sqcap \text{Red}(f) \in \text{Fix}(f)$$

where

$$\begin{aligned} \text{Fix}(f) &= \{a \in D \mid f(a) = a\} \\ \text{Red}(f) &= \{a \in D \mid f(a) \sqsubseteq a\} \end{aligned}$$

Lemma A.2.2 *Let D be a complete lattice and $h: D \rightarrow D$ be a monotone and extensive function. For $x, i \in D$, if $h(x) \sqsubseteq x$ and $i \sqsubseteq x$ then $\text{LFP}_{\sqsupseteq i}(h) \sqsubseteq x$.*

Proof: By definition of extensive function, we get that $x \sqsubseteq h(x)$, and together with the assumption that $h(x) \sqsubseteq x$, we get that $h(x) = x$. Since $i \sqsubseteq x$, we conclude that $\text{LFP}_{\sqsupseteq i}(h) \sqsubseteq x$.

Lemma A.2.3 *Let D be a complete lattice and $h: D \rightarrow D$ be a monotone and extensive function. Given $i \in D$, let $h': D \rightarrow D$ be defined by $h'(x) = h(x) \sqcup i$, for all $x \in D$. Then, $\text{LFP}_{\sqsupseteq i}(h) = \text{LFP}(h')$.*

Proof: Let x be $\text{LFP}_{\sqsupseteq i}(h)$. By definition of $\text{LFP}_{\sqsupseteq i}$, $h(x) = x$ and $i \sqsubseteq x$, and we get that $h(x) \sqcup i = x \sqcup i = x$, that is, $h'(x) = x$. By Tarski, from $h'(x) = x$ we get that $\text{LFP}(h') \sqsubseteq x$. Since $x = \text{LFP}_{\sqsupseteq i}(h)$ by definition, we get that $\text{LFP}(h') \sqsubseteq \text{LFP}_{\sqsupseteq i}(h)$.

Let y be $\text{LFP}(h')$. By definition of LFP , $h'(y) = y$, and using the definition of h' we get that $h(y) \sqcup i = y$. It implies that $i \sqsubseteq y$ and $h(y) \sqsubseteq y$. Using Lemma A.2.2 we get that $\text{LFP}_{\sqsupseteq i}(h) \sqsubseteq y$. Since $y = \text{LFP}(h')$, we get that $\text{LFP}_{\sqsupseteq i}(h) \sqsubseteq \text{LFP}(h')$.

Lemma A.2.4 *Let D be a complete lattice and $f: D \rightarrow D$ be a monotone and extensive function. Let $f^\natural: \mathcal{A} \rightarrow \mathcal{A}$ be defined by $f^\natural = \alpha \circ f \circ \gamma$. Let a^\natural denote $\text{LFP}_{\sqsupseteq \alpha(I)}(f^\natural)$. If $\alpha(T) \sqsubseteq \alpha(I)$, then $\alpha(\text{LFP}_{\sqsupseteq T}(f)) \sqsubseteq a^\natural$.*

Proof: Because a^\natural is a fixed point of f^\natural , we get that $f^\natural(a^\natural) = a^\natural$ and after applying γ we get that $\gamma(f^\natural(a^\natural)) = \gamma(a^\natural)$. From the definition of f^\natural , we get that $\gamma(f^\natural(a^\natural)) = \gamma(\alpha(f(\gamma(a^\natural))))$. By properties of Galois connection, we get that $f(\gamma(a^\natural)) \sqsubseteq \gamma(\alpha(f(\gamma(a^\natural))))$. Therefore, $f(\gamma(a^\natural)) \sqsubseteq \gamma(a^\natural)$.

Since a^\natural is $\text{LFP}_{\sqsupseteq \alpha(I)}$, we get that $\alpha(I) \sqsubseteq a^\natural$. From $\alpha(T) \sqsubseteq \alpha(I)$, we get that $\alpha(T) \sqsubseteq a^\natural$. From monotonicity of γ , we get that $\gamma(\alpha(T)) \sqsubseteq \gamma(a^\natural)$ and from properties of Galois connection we get that $T \sqsubseteq \gamma(\alpha(T))$. Therefore, $T \sqsubseteq \gamma(a^\natural)$.

From $f(\gamma(a^\natural)) \sqsubseteq \gamma(a^\natural)$ and $T \sqsubseteq \gamma(a^\natural)$, using Lemma A.2.2, we get that $\text{LFP}_{\sqsupseteq T}(f) \sqsubseteq \gamma(a^\natural)$. By monotonicity of α and Galois connection, we get that $\alpha(\text{LFP}_{\sqsupseteq T}(f)) \sqsubseteq a^\natural$.

Theorem 2.3.1(Soundness) *If $a \in \mathcal{A}$ is invariant under P and $I \sqsubseteq \gamma(a)$ then a is a sound overapproximation of P .*

Proof: By definition of invariant under P and the function f from Section 2.3.1, f is monotone and extensive and the following holds: $f(\gamma(a)) \sqsubseteq \gamma(a)$. Using Lemma A.2.2, we get that $\text{LFP}_{\sqsupseteq I}(f) \sqsubseteq \gamma(a)$, i.e., a is a sound approximation of P .

Theorem 2.3.2 *Let $f^\natural: \mathcal{A} \rightarrow \mathcal{A}$ be defined by $f^\natural = \alpha \circ f \circ \gamma$. The procedure in Fig. 2.3 computes the least fixed point of f^\natural w.r.t. $\alpha(I)$.*

Proof: Let a^\natural denotes $\text{LFP}_{\sqsupseteq \alpha(I)}(f^\natural)$. Recall that f is monotone and extensive.

Let T_i and a_i denote the set of concrete states T and the abstract value a , respectively, in iteration i of the algorithm. Initially, $a_0 = \perp$ and $\alpha(T_0) = \alpha(I)$. For $i \geq 0$,

$$\begin{aligned} a_{i+1} &\sqsubseteq a_i \sqcup \alpha(\text{LFP}_{\sqsupseteq T_i}(f)) \\ a_{i+1} &\sqsupseteq a_i \sqcup \alpha(T_i) \\ T_{i+1} &= \{\sigma\} \text{ such that } \sigma \in f(\gamma(a_{i+1})) \text{ and } \sigma \notin a_{i+1} \end{aligned}$$

Assume that the procedure terminates after n iterations. From the termination condition follows that $f(a_n) \sqsubseteq a_n$.

First, we prove that upon termination of the procedure, $a^\natural \sqsubseteq a_n$.¹ Initially, $\alpha(I) \sqsubseteq a_1$ because $a_0 \sqcup \alpha(T_0) = \alpha(T_0) = \alpha(I)$. In each iteration $a_i \sqsubseteq a_{i+1}$ by construction in line [6], thus $\alpha(I) \sqsubseteq a_n$. Using the termination condition of the loop we get that $f(a) \sqsubseteq a$. By Lemma A.2.2, we get that $\text{LFP}_{\sqsupseteq \alpha(I)}(f^\natural) \sqsubseteq a_n$, that is, $a^\natural \sqsubseteq a_n$.

Second, we prove that for all $i \geq 0$, $a_i \sqsubseteq a^\natural$. The base case: initially, $\alpha(T_0) = \alpha(I)$ and $\alpha(I) \sqsubseteq a^\natural$, by definition of a^\natural . Using Lemma A.2.4 we get that $\alpha(\text{LFP}_{\sqsupseteq T_0}(f)) \sqsubseteq a^\natural$. Recall that $a_1 = \alpha(\text{LFP}_{\sqsupseteq T_0}(f)) \sqcup \perp$. Thus, $a_1 \sqsubseteq a^\natural$.

By inductive hypothesis, $a_i \sqsubseteq a^\natural$. To prove that $a_{i+1} \sqsubseteq a^\natural$, it is sufficient to show that $\alpha(\text{LFP}_{\sqsupseteq T_i}(f)) \sqsubseteq a^\natural$, because $\text{Execute}(f, T_i) \sqsubseteq \text{LFP}_{\sqsupseteq T_i}(f)$. Recall that $T_i \sqsubseteq f(\gamma(a_i))$. By (2.1) we get that $\alpha(T_i) \sqsubseteq \alpha(f(\gamma(a_i))) = f^\natural(a_i)$. From the inductive hypothesis $a_i \sqsubseteq a^\natural$ and the fact that f^\natural is monotone, we get that $f^\natural(a_i) \sqsubseteq f^\natural(a^\natural) = a^\natural$, because a^\natural is the least fixed point of f^\natural . Thus, $\alpha(T_i) \sqsubseteq a^\natural$, and by Lemma A.2.4 we get that $\alpha(\text{LFP}_{\sqsupseteq T_i}(f)) \sqsubseteq a^\natural$.

Theorem 2.3.3 *If the lattice \mathcal{A} has a finite height, then the procedure in Fig. 2.3 terminates.*

Proof: Consider a concrete state σ chosen in the i -th iteration of the procedure. Recall that Execute is guaranteed to terminate. From the properties of Execute , it follows that $\sigma \in T$. Using the fact that join distributes over α , we get that $\alpha(C) = \sqcup_{c \in C} \alpha(c)$, and we can write line [6] as $a_{i+1} = a_i \sqcup \alpha(\{\sigma\}) \sqcup \sqcup_{C \setminus \{\sigma\}} \alpha(C)$. From line [7] follows that $\sigma \notin \gamma(a_i)$. Therefore, $a_i \sqcup \alpha(\{\sigma\})$ is strictly higher than a_i in the abstract lattice \mathcal{A} .

¹It also implies the soundness of the procedure.

Appendix B

Proofs for Chapter 3

Lemma B.0.5 *Consider the content of the set $result$ at the end of $\hat{\alpha}$ procedure. If $S \in result$ then there exists S^{\natural} such that $S^{\natural} \models \varphi$ and $\beta(S^{\natural}) = S$.*

Proof: For the sake of argument, assume that there exists $S \in result$ such that for all concrete structures S^{\natural} that satisfy φ and embed into S , $\beta(S^{\natural}) \neq S$.

Recall that at the end of *bif* procedure, all the abstraction predicates have definite values in S . During phase 2, relation values can only be lowered, meaning that the abstraction predicates remain definite. Consequently, if S^{\natural} is embedded into S , then $\beta(S^{\natural})$ is embedded into S using the identity function, because embedding preserves canonical names; $\beta(S^{\natural})$ embeds into S by an identity function only when the set of canonical names in S^{\natural} and S is the same. Therefore, the assumption $\beta(S^{\natural}) \neq S$ implies that there exists a relation whose value in S is indefinite, but in $\beta(S^{\natural})$ it is definite.

Formally, for each concrete structure that satisfies φ and embeds into S , there exists a relation q with an indefinite value on some tuple of nodes u_1, \dots, u_k in S , such that the value of q on all tuples of nodes in the concrete structure S^{\natural} that are mapped to u_1, \dots, u_k by the embedding, is the same.

In phase 2 of $\hat{\alpha}$ procedure, when the value of q on u_1, \dots, u_k in S is examined, the first if-condition is true, because the formula $\hat{\gamma}(S) \wedge \varphi \wedge \varphi_{q, u_1, \dots, u_k}$ is not satisfiable, as follows from the assumption. Therefore, the statement guarded by this if-condition is executed, removing the structure S from *result* set. Therefore, a contradiction is obtained.

Lemma B.0.6 *For each structure $S \in result$, there exists a concrete structure S^{\natural} that satisfies φ and embeds into S .*

Proof: By induction on the steps of $\hat{\alpha}$ procedure. At the end of *bif* procedure, this holds due to Lemma B.0.7. Each iteration of the main loop in $\hat{\alpha}$ preserves this, because structure S_0 or S_1 can be added to *result* only when the if-condition that guards its statement is true. The if-condition requires that there exists a concrete structure that satisfies φ and embeds into the structure to be added to *result*.

Lemma B.0.7 *At the end of *bif* procedure, each structure in X represents at least one concrete structure that satisfies φ .*

Proof: After checking the precondition, all structures in X the claim holds. When a structure is added to X , there are three cases to consider.

First, if S' is added to X . In this case, there exists a concrete structure that satisfies $\hat{\gamma}(S) \wedge \varphi \wedge \varphi_{q, u}$, denote it by S^{\natural} . Consequently, S^{\natural} satisfies φ and embedded into S . Using Lemma B.0.8, S^{\natural} is embedded into S' , proving the claim.

In the second case, S_0 is added to X in statement $X := X \cup \{S_0\}$. This statement is executed when the if-condition that guards it is true, i.e, there exists a concrete structure that satisfies $\hat{\gamma}(S_0) \wedge \varphi$. In particular, this concrete structure is represented by S_0 and satisfies φ , proving the claim. The third case, in which S_1 is added to X , is symmetric to this case.

Lemma B.0.8 *Consider an iteration of the while-loop in *bif* procedure. Let $S \in W$, q be an abstract predicate and $u \in U^S$ handled in that iteration. Let S^{\natural} be a concrete structure such that $S^{\natural} \models \varphi$ and S^{\natural} is embedded into S . S^{\natural} is embedded into one of the structures $\{S', S_0, S_1\}$, denote it by S'' .*

Proof: By assumption, there exists embedding function f such that $S^{\natural} \sqsubseteq_f S$. Show that there exists $S'' \in \{S', S_0, S_1\}$ such that S^{\natural} is embedded into S'' by constructing an embedding function $f': S^{\natural} \mapsto S''$, based on f .

- If $S^{\natural} \models \widehat{\gamma}(S) \wedge \varphi \wedge \varphi_{q,u}$ then S^{\natural} contains two nodes, denoted by u_0 and u_1 , such that the value of q on u_0 is 0 and the value of q on u_1 is 1. In this case, S^{\natural} is embedded into S' using the following embedding function:

$$f'(u^{\natural}) = \begin{cases} u.0 & \text{if } f(u^{\natural}) = u \text{ and } \iota^{S^{\natural}}(q)(u) = 0 \\ u.1 & \text{if } f(u^{\natural}) = u \text{ and } \iota^{S^{\natural}}(q)(u) = 1 \\ f(u^{\natural}) & \text{otherwise} \end{cases}$$

f' is well-formed because f is and $\iota^{S^{\natural}}(q)(u)$ cannot be 0 and 1 simultaneously. f' is surjective: its image includes $u.0$ and $u.1$, because $f'(u_0) = u.0$ and $f'(u_1) = u.1$ as follows for the denotations above; other elements of S' are images of f' , because f' is the same as f and f is surjective, by assumption.

Show that f' preserves relation values. The values of all relations on all tuples in S' , are the same as in S , except the value of q on the new nodes $u.0$ and $u.1$. f' preserves these values, because f does and f' is the same as f for the relevant nodes (these are the concrete nodes in S^{\natural} that are **not** mapped to the new nodes of S'). Let $u^{\natural} \in S^{\natural}$ such that $f(u^{\natural}) = u$. Without loss of generality, let $\iota^{S^{\natural}}(q)(u^{\natural}) = 0$. By definition of f' , $f'(u^{\natural}) = u.0$. By definition of S' , the value of q on $u.0$ is 0, that is the same as the value of q on u^{\natural} . It shows that f' preserves the values of q . The case where $\iota^{S^{\natural}}(q)(u^{\natural}) = 1$ is symmetric.

- If S^{\natural} does not satisfy $\widehat{\gamma}(S) \wedge \varphi \wedge \varphi_{q,u}$ then the value of q on all nodes in S^{\natural} that are mapped to u by the embedding is the same. If this value is 0, S is embedded into S_0 , otherwise — into S_1 . These cases are symmetric, therefore we consider only the former. Note that S and S_0 have the same universe, and differ only in the value of q on u . Hence, the embedding function $f': S^{\natural} \mapsto S_0$ is the same as f . f' is well-formed and surjective because f is. For all relation values, except the value of q on u , f' preserves the values of the relations, because these values are the same in S and S' . The value of q on u in S_0 is 0, by construction of S_0 . The value of q on all nodes in S^{\natural} that are mapped to u by f' is 0, by assumption. Therefore, the value of q is preserved by f' .

Lemma B.0.9 *If $S^{\natural} \models \varphi$ then there exists $S \in \text{result}$ such that $S^{\natural} \sqsubseteq S$.*

Proof: Use induction on the value of *result* at each step of the procedure.

The base case: after the initialization phase $\text{result}_0 = \top$, therefore it represents all concrete structures, in particular all structures that satisfy φ .

The induction step: Let S^{\natural} be a concrete structure such that $S^{\natural} \models \varphi$. Assume that after i steps of the procedure, the hypothesis holds: there exists $S_i \in \text{result}_i$ such that $S^{\natural} \sqsubseteq S_i$. Show that after step $i + 1$, there exists $S_{i+1} \in \text{result}_{i+1}$ such that $S^{\natural} \sqsubseteq S_{i+1}$.

If step $i + 1$ is the call to the procedure $\text{bif}(\text{result})$, the conclusion is obtained from Lemma B.0.8, because all concrete structures satisfying φ that are represented by an abstract structure, are also represented by a bifurcation of the abstract structure — there is no loss of “important” structures during bifurcation.

Otherwise, step $i + 1$ is an operation performed during the inner loop of phase 2. Suppose that it operates with a structure S , relation q of arity k and node tuple u_1, \dots, u_k in S . The only structure that could be removed from *result* in this step is S .

Recall that S_i is the structure in result_i that, by assumption, represents S^{\natural} . If the structure S , that can be removed from *result* is **not** S_i , the hypothesis holds for $i + 1$ and S is the structure that represents S^{\natural} in result_{i+1} , i.e., S_{i+1} is S . Otherwise, S and S_i is the same structure, thus there exists an embedding function g such that $S^{\natural} \sqsubseteq_g S$. We shall prove that if S is removed from *result*, then one of the structures S_0 or S_1 represents S^{\natural} and it is added to *result*.

According to the algorithm, S is removed when all concrete structures represented by S that satisfy φ have the same value for all node tuples mapped to u_1, \dots, u_k by the embedding. In particular, this holds for S^{\natural} . Without loss of generality, assume that the value is 0 and show that S^{\natural} is embedded in S_0 . The embedding function f such that $S^{\natural} \sqsubseteq_f S_0$ is g : (i) because S and S_0 have the same universe, f is well-defined and surjective; (ii) we only

have to show that f preserves values of q over u_1, \dots, u_k , because the values of other relations are the same in S and S_0 . The value of q over all tuples mapped to u_1, \dots, u_k is 0 by assumption, and $\iota^{S_0}(q)(u_1, \dots, u_k)$ is 0 by the construction of S_0 .

To complete the proof, we have to show that S_0 is added to *result*, that is the if-condition that guards the statement $result := result \cup \{S_0\}$ is true. We have to show that there exists a concrete structure that satisfies the formula $\widehat{\gamma}(S_0) \wedge \varphi$. Indeed, S^{\natural} satisfies the condition: S^{\natural} satisfies $\widehat{\gamma}(S)$ because it is embedded into S^{\natural} as shown above; also, by assumption, S^{\natural} satisfies φ .

Recall that $\widehat{\gamma}$ is only defined for bounded structures. The following lemma is a prerequisite for the use of $\widehat{\gamma}$ in the *assume* algorithm. It shows that if *assume* is applied to a bounded structure, then all the structures created by *assume* are bounded, and therefore $\widehat{\gamma}$ can be used.

Lemma B.0.10 *In every step of assume algorithm, the result is a bounded structure, given that the input is a bounded structure.*

Proof: Assume that the input of each operation considered below is a bounded structure. Then, to violate this “boundedness” property, the operation must change a definite value of some abstraction predicate, according to the definition of a bounded structure. (from 1 to 0 or 1/2 and from 0 to 1 or 1/2).

The procedure *bif* either (i) lowers a value of a relation from 1/2 to 1 or 0, or (ii) duplicates a node and sets an abstraction predicate q with indefinite value to definite values on the two copies of the node. Both operations do not violate “boundedness” property. Also, phase 2 of the algorithm by its definition can only lower relation values, therefore it cannot violate the “boundedness” property.