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

PROPERTY-GUIDED VERIFICATION OF CONCURRENT HEAP-MANIPULATING PROGRAMS

by

Eran Yahav

under the supervision of Dr. Mooly Sagiv

A thesis submitted
for the degree of Doctor of Philosophy

Submitted to the Senate of Tel-Aviv University
October 2004

To my parents.
To my dearest ones, Michal, Jonathan, Yaron,
and future number 3.

Abstract

Property-Guided Verification of Concurrent Heap-Manipulating Programs

Eran Yahav
Doctor of Philosophy
School of Computer Science
Tel-Aviv University

We address the problem of verifying properties of concurrent and sequential programs written in languages, such as Java, that make extensive use of the heap to allocate—and deallocate—new objects and threads. We present a framework for the verification of sequential and concurrent Java programs. The framework combines thread scheduling information and information about the shape of the heap. This leads to error-detection algorithms that are more precise than existing techniques. In contrast to many existing verification techniques, our algorithms do not put a bound on the number of allocated objects (and threads). We also present novel approaches that allow us to tie the cost of verification to the nature of the property being verified. The combination of these techniques allows us to automatically verify non-trivial properties of heap-manipulating programs that have not been automatically verified in the past.

Acknowledgements

I have been fortunate to have Mooly Sagiv as my advisor. His knowledge, guidance, optimism, enthusiasm, and above all—patience, were crucial in every step towards the completion of this thesis.

I would like to thank Ramalingam and John Field for two enjoyable and extremely fruitful summers I spent at the IBM T.J. Watson Research Lab.

I would like to thank Tom Reps and Reinhard Wilhelm for their guidance, help, and support throughout the course of this thesis. Working with them has been a real privilege.

I would like to thank Ran Shaham for many fruitful discussions, and for many insights that made this work better. I also want to thank him for taking the time and effort of reading earlier drafts of this thesis.

I would like to thank Roman Manevich for many discussions and for his endless efforts to provide a solid experimental analysis framework. This work would have been impossible without his help and support.

I would also like to thank Nurit Dor and Noam Rinetzky for reading earlier drafts of this thesis.

Special thanks to David Oren for patiently reviewing earlier drafts of most thesis chapters.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Concurrent programming is becoming a common practice in modern software development. A concurrent program allows a number of activities to be performed together. This enables the programmer to increase the availability and reactivity of his program, for example by allowing the user to perform other activities while a long computation takes place concurrently.

The Java language brought concurrent programming to a wide-range of products and programmers due to its natural support of multi-threading (a multithreaded program is a program which concurrently executes a number of threads where each *thread* may be viewed as a single sequential program). Java concurrency is quite common, it can be found in commercial applications such as web-servers, Java applets, multimedia applications and others. However, despite its being widespread, writing a correct concurrent program is as hard and error-prone as ever.

While concurrent programming introduces additional strength in the design and implementation of software systems, it also introduces problems that do not emerge in sequential programs (e.g., data-races and deadlocks).

Debugging and testing of a concurrent program is a complicated task since the results of an execution may depend on the specific order in which threads are scheduled. Furthermore, the concurrency model used by modern programming languages, such as Java, provides low-level concurrency-control constructs that enable the programmer to create complicated and powerful synchronization schemes.

Although it is well known that programs using these concurrency models are hard to debug, existing programming environments provide no compile-time support for checking the correctness of concurrent behavior. For example, the Java language provides no means for compile-time checking and almost no means for runtime checking of the correctness of concurrent behavior. This makes concurrent programming in Java quite error-prone (e.g., [108, 51]).

Generally, one would like to prove that a given concurrent program is correct with respect to some specification with the same certainty one proves a mathematical theorem. This is the essence of program verification.

Ideally, the process of verification would be a fully automatic process taking a program and a specification as input and supplying a "yes/no" answer as to whether the program satisfies the specification. Unfortunately, this problem is known to be undecidable, that is, it is possible to show that an automatic solution cannot exist for arbitrary programs and specifications. Moreover, it is very difficult to specify the full behavior of real software.

This research focuses on the use of sound program analysis techniques for the verification of concurrent Java programs. Once convinced that automatic verification is a subtle problem, one may wonder why we believe that we can provide a reasonable solution. The main reason is that we do not attempt to tackle the general problem of automatic verification, but rather focus on the following:

(a) using sound approximation of program behavior — use a static (compile-time) approximation of all possible behaviors of a program. The approximation contains all possible behaviors of the program, but may also contain some superfluous behaviors. Verifying the property against the approximated behavior can only err on the safe side, that is, it detects all possible errors (since all "real" behaviors are included), but may produce *false alarms* due to the superfluous behaviors included. That is, our algorithms never miss an error but there are cases in which the property is reported to be violated, where it is not violated in any real execution of the program. It is challenging to develop algorithms that yield a tolerable number of false alarms.

(b) Concentrating on specific properties and not addressing full program correctness. This goes both for the specification and the verification algorithm.

(c) Possibly safely omitting some aspects of program semantics (e.g., arithmetics).

(d) Considering a specific model of concurrency — the Java concurrency model which uses threads communicating through shared-memory.

Two of the main challenges of software verification are handling heap-allocated storage and handling dynamic allocation of objects and threads. These features are often ignored or handled in an imprecise manner by existing verification and static-analysis approaches, especially for concurrent programs [87].

One of the main problems addressed by our research is therefore:

Problem 1 *[Feasibility] Given a non-trivial concurrent Java program with dynamic allocation of objects and threads, and a non-trivial property, is it possible to automatically verify that the program satisfies the property of interest, while producing a tolerable number of false alarms?*

Finite-state verification techniques have been successfully applied in the verification of hardware systems and protocols (e.g., [17]). However, they cannot be immediately applied for verification of Java programs, which support dynamic allocation of objects and threads with no a priori bound. The problem is that Java supports both unbounded data structures and unbounded control structures (dynamic allocation of objects and threads).

One of the promising approaches for automatic verification of software is to perform model-checking (exhaustive state-space exploration) of an abstract finite-state model of the system at hand. A conservative abstraction of the original system is used to simulate its behavior assuring that any property established to hold for the abstract model is guaranteed to hold for the original system. The problem now becomes a problem of finding the "right" abstraction mapping, an abstraction that maintains the necessary observations for verifying the property of interest, and abstracts away non-essential information to make verification feasible.

Abstract interpretation [25] traditionally uses an abstraction mapping that is defined for some analysis, and is independent of the specific property of interest. This allows abstract interpretation to be applied automatically and uniformly to any arbitrary program. It allows the designer of the analysis to develop a clever and complicated representation when the analysis is designed. However, an analysis targeted at a wide range of properties has the major disadvantage of tying the cost of verification to the finest property that should be *observable*. That is, verification of a simple property may be as expensive as verification of a complex property. Moreover, it could lead to a non-tolerable number of false alarms when the domain is inappropriate for the property of interest. One of the challenges in that respect is therefore deriving a specialized program analysis algorithm that is only as precise (and as expensive) as needed. That is, how to use a given property specification to derive a program analysis algorithm in a way that the property of interest remains *observable* while as many other details of the original program (irrelevant for verification of the property of interest) are abstracted away. This raises the second problem addressed by this research:

Problem 2 *[Property-Guidedness]*

(a) ***Construction*** *Given a non-trivial property, automatically construct a program analysis such that the given property is* observable *for a set of programs. We refer to such construction as being property-guided.*

(b) ***Refinement*** *Given a non-trivial property, and a program analysis algorithm PA, how can the property be used to refine the abstraction applied by PA.*

1.1 Thesis Contributions

In this section, we give a brief description of the main contributions of this thesis.

Specification

We have defined a specification language called *Evolution Temporal Logic* (ETL) for defining requirements on program behavior addressing both time and space. Unlike classical model checking, which uses propositional temporal logic, we use a first-order temporal logic to specify temporal properties of

heap evolutions; this logic allows domain changes to be expressed, which permits allocation and deallocation to be modelled naturally [119]. ETL and its verification algorithms are described in Chapters 4 and 5.

Automatic Verification

We have defined several algorithms for verifying ETL specification for concurrent Java-like programs [115, 119, 95, 117, 116]. Our algorithms are able to handle both safety and liveness properties specified as ETL formulae. Our algorithms combine thread-scheduling information and information about the shape of the heap. This leads to error-detection algorithms that are more precise than existing techniques. In contrast with existing verification techniques, our algorithms do not put a bound on the number of allocated objects (and threads). A basic algorithm for verifying non-temporal safety properties for concurrent programs is described in Chapter 2. Chapter 4 and Chapter 5 describe algorithms for verification of general ETL properties.

Property-Guided Abstraction

We have also investigated several techniques for guiding the abstraction by the property being verified, resulting in more efficient and possibly more precise verification algorithms. In particular, we defined techniques for automatic predicate derivation described in Chapter 3, and for property-based separation and heterogenous abstraction described in Chapter 6.

One of the primary intuitions behind the algorithms presented in this part of the thesis is that maintaining just the right correlation required between "analysis facts" can be the key to efficient and precise verification: maintaining no correlations (independent attribute analysis) can lead to imprecision, while maintaining all correlations (relational analysis) can lead to inefficiency.

Applications

The combination of the above techniques allows us to successfully verify non-trivial properties of concurrent and sequential heap-manipulating programs such as implementations of concurrent queue algorithms [120]. In particular, we verified partial correctness of the two-lock queue algorithm that is part of the `java.util.concurrent` package of JDK1.5. We have also used these techniques to establish temporal properties for compile-time memory management [95], and solve verification challenges such as the apprentice challenge presented by J. Moore [73]. Applications of our techniques are described in Chapter 7.

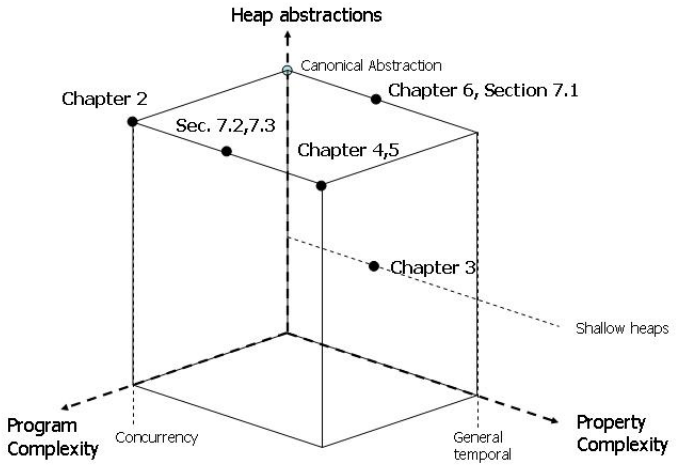


Figure 1.1: Overview of thesis chapters.

1.2 How to Read this Thesis

The contributions of this thesis are both theoretical and practical. This section gives an outline of thesis chapters and describes two possible paths of going through the presented material.

In Chapter 2, we describe a parametric framework for verifying safety properties of concurrent Java programs; this partly answers Problem 1. A preliminary version of this work appeared in [115].

In Chapter 3, we consider the problem of *typestate verification* for *shallow programs*; i.e., programs where pointers from program variables to heap-allocated objects are allowed, but where heap-allocated objects may not themselves contain pointer fields. In this chapter we show how to construct a property-guided abstraction for shallow programs and certain classes of properties. This chapter partly addresses Problem 2(a). A preliminary version of this work was published in [41].

Chapter 4 and Chapter 5 show two verification approaches for verifying general temporal properties of heap-manipulating programs. The abstraction used in this chapter is refined by the verified property, thus partly addressing Problem 2(b). Preliminary versions of part of this work appeared in [119] and in [116].

In Chapter 6, we show how *separation* (decomposing a verification problem into a collection of verification subproblems) can be used to improve the efficiency and precision of verification of safety properties. A preliminary version of this work was published in [117].

Chapter 7 describes applications of our framework for verifying various properties of concurrent and sequential heap-manipulating programs. Preliminary versions of results summarized in this chapter appeared in [95], [120], and [114].

Fig. 1.1 shows a classification of thesis chapters over a 3-dimensional cube. The dimensions of this cube are:

Heap Abstraction Describes the strength of the applied heap-abstraction. Zero on this axis means that no heap abstraction is used, thus forcing an assumed a priori bound on the number of allocated objects and threads. The heap abstraction used in most chapters of this thesis is the canonical abstraction, described in Chapter 2.

Program Complexity Describes the complexity of the programs that could be handled. Along this dimension we only distinguish between sequential and concurrent programs. Our treatment of concurrent programs is described in Chapter 2. To simplify presentation, the material in the rest of the chapters is mostly presented in terms of sequential programs.

Property Complexity Describes the complexity of the properties that could be handled. Property complexity ranges from non-temporal safety properties (e.g., as used in Chapter 2) to full temporal specification that support specification of liveness properties (e.g., Chapter 4).

The thesis could be read following a theoretical or a more practical track. Readers more interested in the theoretical contributions of this thesis should read Chapter 2, Chapter 3, Chapter 4, and Chapter 5. Readers more interested in practical contributions should read Chapter 2, Chapter 6, and Chapter 7.

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. The section is organized as follows. Section 1.3.1 describes the various specification languages used in this thesis. Section 1.3.2 contrasts our integrated verification approach (integrating verification and pointer-analysis) with the common two-phased approach, providing some intuition to its preferable precision, and showing how the abstraction is refined by the property being verified (in the spirit of Problem 2(b)). In Section 1.3.3 we show how specialized abstractions are constructed from user specifications (in the spirit of Problem 2(a)). Finally, in Section 1.3.4 we discuss verification of temporal properties.

1.3.1 Specification

In this thesis, we use various specification languages for describing correctness properties of concurrent and sequential heap-manipulating programs. Our choice of specification language in each chapter is aimed to simplify presentation and describe the key concepts of the chapter with minimal specification clutter.

	Safety	**Liveness**
Single object	*Typestate* finite automata (regular expressions) ETL	Evolution Temporal Logic (ETL)
Correlated objects	*First-Order Safety* `Easl` FO^{TC} (non temporal) ETL	

Figure 1.2: Specification languages used in this thesis classified by the kind of properties they describe.

The specification languages we use range from first-order logic with transitive closure (FO^{TC}), through regular-expressions, and up to Evolution Temporal Logic (ETL) which is essentially a first-order linear temporal logic. Fig. 1.2 shows the specification languages used in this thesis, classified by the kind of properties they describe. The dimensions of the table are:

Safety / Liveness A property may be classified as a safety property that requires that nothing "bad" ever happens, or as a liveness property that requires that something "good" eventually happens. In this thesis, we address both safety and liveness properties.

Single object / Correlated objects A property may require a correct behavior of each object independently, or involve multiple correlated objects. When the correctness of a property may be verified for a program object independently of other program objects, we classify the property as a *single object* property. When the property involves multiple correlated objects, we classify it as a *correlated objects* property. Single object safety properties are often referred to as *typestate* properties [103]. Multiple object safety properties are referred to as *first-order safety* properties [84].

As an example, consider a File object that has two possible states: $open$ and $closed$; supports two operations: `read()`, and `close()`; and assumed to be in its $open$ state when created.

For this File component, we would like to specify the safety property:

a file is not read after it has been closed. (1.1)

Since this property considers each file separately (and independently of other files), and requires that nothing "bad" (read after close) ever happens, it is classified as a single-object safety property (also referred to as a typestate property).

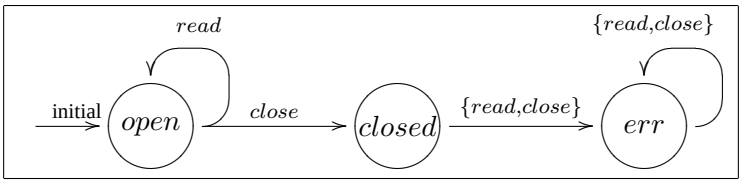


Figure 1.3: A finite state automaton for the property `read`*;`close`.

Specification as a Non-Temporal Safety Property

A non-temporal safety property is a property that is evaluated in every global state of the program (configuration) independently of other global states.

Property 1.1 could be specified as the following non-temporal safety property:

$$\forall v.\neg(closed(v) \wedge read(v))$$

assuming that a unary predicate $closed(v)$ is set to hold for a file v when the file is closed, and a predicate $read(v)$ is set to hold for a file v when the file is being read from.

In Chapter 2, we use FO^{TC} formulae to describe non-temporal safety properties.

Specification via Regular Expressions

A safety property that specifies the behavior of each object independently of other objects (typestate property) could be specified using a regular-expression, or equivalently by using a finite-automaton. Technically, the regular-expression is taken to be a universally quantified specification that should hold for all objects of the specified type.

Property 1.1 could be also specified as a simple regular expression observing the events of invoking `read()`, and `close()`:

`read`*;`close`

When specifying a safety property using a regular expression, we adopt the convention that a regular expression α denotes the *prefix closure* of the set of sequences of operations defined by α. For example, when we write `read`*;`close` we also consider ϵ (the empty sequence) and `read` to be valid sequences. This property could be equivalently specified using the finite automaton in Fig. 1.3[1].

Chapter 3 uses finite-automata and regular expressions for describing typestate properties that put more emphasis on the sequence of events that may occur for an object in an execution of the program. Finite-automata are also used to specify memory-management properties in Section 7.1.

[1]since all states but err are accepting we do not mark accepting states in the figure.

Specification via ETL

Property 1.1 involves two events, closing a file, and reading from the file. While specification of this property as a non-temporal safety property makes implicit assumption on how the predicates $read(v)$ and $closed(v)$ are updated, or on when an error is reported (e.g., should an error be reported on close after read?), specification via ETL exposes the legal sequences of events. The property

$$\Box(\forall v.close(v) \rightarrow \Box\, \neg read(v))$$

explicitly states that when an object has just been closed, it should not be read from in the future. Intuitively, the meaning of the *globally* temporal operator applied to an ETL formula φ (denoted by $\Box\, \varphi$) is that φ should hold forever from the current point of computation.

Chapter 4 uses Evolution Temporal Logic (ETL), a general specification language, to describe general temporal properties of heap-manipulating programs.

In order to specify liveness properties, this thesis only uses ETL specifications (although in principle we could have used Büchi-automata for subclasses of ETL, e.g. as in [106]). For example, we could write a property that requires that a reference `f` refers to an open file object infinitely often:

$$\Box \Diamond \exists v.f(v) \wedge \neg closed(v) \tag{1.2}$$

Where the temporal operator *eventually* applied to an ETL formula φ (denoted by $\Diamond\, \varphi$) intuitively means that there exists a point in the future of the computation in which φ holds, and $f(v)$ is a unary predicate representing the fact that the object v is pointed to by the reference variable `f`.

While ETL is a general specification language, it is often more convenient to use alternative specification methods such as finite-automata or the `Easl` specification language (see below) when only interested in specifying safety properties.

Specification via `Easl`

`Easl` [84] is a procedural specification language that can be used for specifying an abstract semantics for a component (or a set of components). `Easl` statements are a subset of Java statements containing assignments, conditionals, looping constructs, and object allocation. `Easl` types are restricted to booleans, heap-references, and built-in abstract Set and Map types. Finally, `Easl` provides a `requires` statement to specify the correct usage constraints imposed by the component: it is the responsibility of any program that uses the component to ensure that the condition specified by the `requires` clause will hold at the corresponding program point. `Easl` supports object references and dynamic allocation. This allows us to naturally express the structural relationships between the objects of interest, as well as dynamic allocation of these objects.

Property 1.1 could be alternatively specified using the `Easl` specification language, as shown in Fig. 1.4. In this specification, the state of a file object is modeled using a boolean field `closed` which

```
class File {
  boolean closed;
  File() {
    closed = false;
  }
  void read() {
    requires !closed;
  }
  void close() {
    closed = true;
  }
}
```

Figure 1.4: An `Easl` specification for the property $\texttt{read}^*;\texttt{close}$.

is initially set to false. Invoking `close()` on a file object sets the `closed` field to true, and invoking `read()` requires that the file has not been closed (i.e., that the value of `closed` is false).

Chapter 6 uses the `Easl` specification language, that can express first-order safety properties (corresponding to the subclass of universally quantified safety properties in ETL).

1.3.2 Integrated Verification and Property-Guided Abstraction

A common approach to verification of heap-manipulating programs is to break the verification problem into two phases: (i) a preprocessing phase in which a conservative finite approximation of the heap is computed; (ii) a verification phase using the finite representation produced by the preprocessing phase.

This approach (referred to as the *two-phased approach*) is used in most existing verification frameworks (e.g., [27, 7, 19, 36]). One of its advantages is the fact that it allows the first phase to be performed in a flow-insensitive manner, which is more scalable. However, while this approach is appealing due to its simplicity and sometimes scalability, it may result in a significant loss of precision, and produce a large number of false alarms. Moreover, in some cases, the loss of precision results with inefficiency due to the exploration of a large number of superfluous states.

The verification algorithms described in this thesis have a common theme of performing an integrated verification and pointer analysis. We refer to this approach as the *integrated approach*.

Generally, the analysis of combined abstract domains (e.g., our integrated approach) is more precise than the combination of separate analyses of abstract domains (e.g., the two-phased approach) [25]. In particular, in this section we demonstrate that even when using a rather limited (and scalable) points-to analysis (in contrast to the shape analysis used in later sections) it may be profitable to use an integrated analysis.

```
[1]  while (...) {
[2]    f = new File();
[3]    f.read();
[4]    f.close();
[5]    // do something
     }
```

Figure 1.5: A simple example program in which loss of precision in the two-phased approach leads to a false alarm.

Consider the code snippet in Fig. 1.5. A new File component is allocated and used in every loop iteration. For this program, we would like to show that a file is never read after it has been closed.

Two-phased Approach

Fig. 1.6 shows the result of applying the two-phased approach for the purpose of verifying that a File component is never read after it has been closed.

We omit the information for lines 1 and 2, since the interesting program points are the ones immediately after allocation, and immediately after closing the file. We assume that the preprocessing phase applies a points-to algorithm based on an allocation-site abstract domain (e.g., [39]). The column *Pointer Analysis Phase* shows the results of the pointer analysis. This could be computed in a flow-insensitive manner.

In this thesis, we depict heap configurations as directed graphs. A node in the graph represents a heap-allocated object. Nodes that potentially represent more than a single heap allocated objects are called *summary nodes* and are depicted as nodes with double line boundaries. Properties of objects are represented using edges from property symbol to the object node. Dashed edges in the graph represent *may* information. For example, the result of pointer analysis at 3 is a single summary node (abstract object). All heap allocated objects summarized by this summary node are allocated at 2, as represented by the solid edge from $site[2]$. The dashed edge from `f` to this summary node represents the fact that it may be pointed to by `f` at this program point.

These notions will be formalized in future chapters.

In the finite-verification flow-sensitive phase, we start with the pointer information at line 2 and initialize the state of the File component to be open (non-closed). Then, at line 3, the file referenced by `f` is read, and its state remains unchanged. When interpreting the statement at line 4, the file referenced by `f` should change its state and become closed. However, not all the objects represented by u_1 are necessarily referenced by `f` (in fact, it is clear that at most one object could be referenced by `f` at any given program point). Updating the state of u_1 to be closed can therefore be unsound, as it fails to represent some of the possible program states. To guarantee soundness, the state of u_1 has to be updated to "unknown", meaning that it may be either open or closed. This is depicted by using a dashed edge

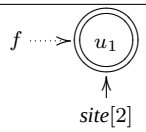
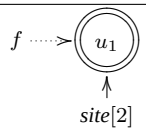
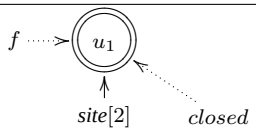
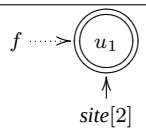
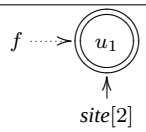
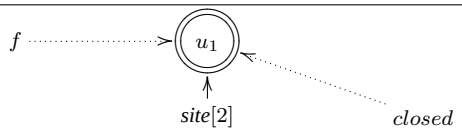
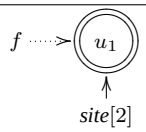
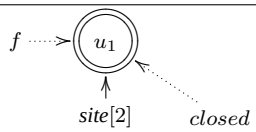
Program Point	Pointer Analysis Phase	Verification Phase	
		1st Iteration	2nd Iteration
[3]			
[4]			 **read may be erroneous**
[5]			

Figure 1.6: Two-phased analysis example.

from $closed$ to u_1 in the figure.

This kind of update to the state of the object is known as a *weak update* ([13]), where the result of the update is a set of possible states, including the old state. Weak updates often produce overconservative results, producing a large number of false alarms.

Next, in the second verification iteration, the allocation at line 2 does not change the possible state for u_1. Therefore, when reaching the invocation of `read()` at 3, the reference `f` may be pointing to a closed file, which causes the verification to produce an error. This reported error is a false alarm since in the program of Fig. 1.5, a file cannot be read after it has been closed.

Integrated Approach

We now show how an integrated approach successfully verifies that a file is not read after it has been closed for the program of Fig. 1.5.

Fig. 1.7 shows the stages of the integrated flow-sensitive verification algorithm. Again, we omit the information for lines 1 and 2. The integrated analysis also uses allocation-site based pointer abstraction, as used in the two-phased approach, but integrating the pointer-analysis with the verification phase allows us to refine the heap-abstraction using the state of the file (closed or open). Thus, objects allocated at the same allocation site but in different states are abstracted to different elements in the abstract domain.

First, at line 3, `f` references an object allocated at line 2 that is in the open (non-closed) state (represented by the absence of an edge from $closed$ to u_1). Then, at line 3, the state of the object remains unchanged. The statement `f.close()` at line 4 has the effect of changing the state of the

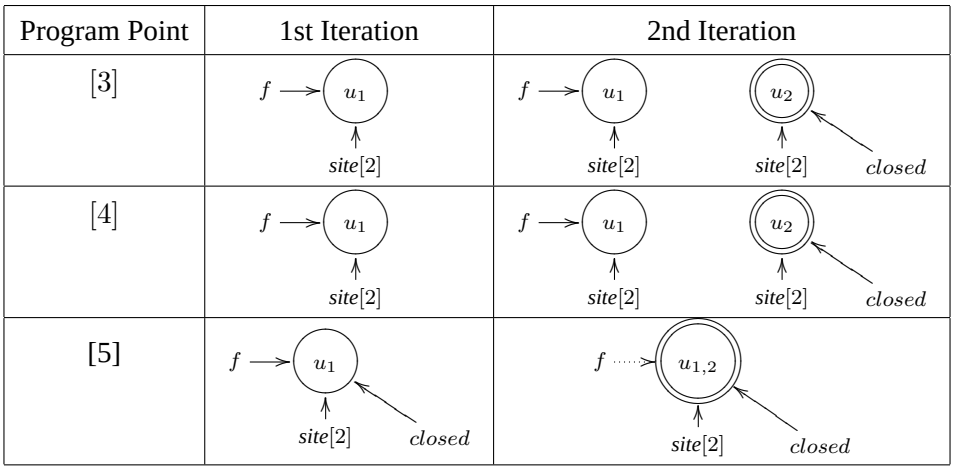


Figure 1.7: Integrated analysis example.

object referenced by `f` to be closed. This kind of update to the state of the object is known as *strong update* ([13]), in which the update results with a new state without the need to record the old state of the object as an alternative.

Next, in the second iteration at line 3, `f` references an object allocated at 2 which is in the open (non-closed) state. This is due to the allocation in 2. The statement at line 3 doesn't change the state, and the the second iteration of `f.close()` at line 4 results with the same abstract state we had in the previous iteration on entry to line 5. At this point of the algorithm, we reach a fixed-point, and no new configurations arise. Since the property is never violated by these abstract configurations, we can conclude that the property holds, and in our example program a file is never read after it has been closed.

Refining the heap abstraction by the state of the component (the File component in the above example) is made possible by integrating the pointer-analysis with the verification phase which maintains the state of the component. This is a special case of property-guided abstraction in which the property being verified is used to refine the heap abstraction. This refinement provides a partial solution to Problem 2(b). In Chapter 4, we will see how temporal properties are used to refine the heap abstraction in a similar manner.

1.3.3 Property Guided Abstraction—Specialized Abstractions

Maintaining just the right correlation required between "analysis facts" can be the key to efficient and precise verification. In this part of the thesis, we derive specialized abstractions by using the specified property and possibly additional guidance provided by the user.

Program Point	1st Iteration	2nd Iteration
[3]	$\langle\{f\}, \{open, closed, err\}\rangle$	$\langle\{f\}, \{open, closed, err\}\rangle$
[4]	$\langle\{f\}, \{open, closed, err\}\rangle$	$\langle\{f\}, \{open, closed, err\}\rangle$
[5]	$\langle\{f\}, \{closed, err\}\rangle$	$\langle\{f\}, \{closed, err\}\rangle$

Figure 1.8: Analysis of the example program using specialized abstraction derived from the property of interest.

Using Specialized Abstraction

Chapter 3 shows how to construct a specialized abstraction for certain kinds of properties and for a restricted kind of programs called *shallow programs*. In shallow programs, pointers from program variables to heap-allocated objects are allowed, but heap-allocated objects may not themselves contain pointers. We use the class of shallow programs to investigate the relation of the complexity of verification to the nature of the property being verified. The idea there is, again, to construct an abstraction that integrates pointer information with information about the state of the verified component. Constructing such specialized abstractions allows us to provide polynomial verification algorithms for certain kinds of properties.

One of the algorithms presented in Chapter 3 can be immediately applied to verify the property of interest in our example program. This algorithm (presented in Section 3.3) can be used to derive a polynomial number of predicates of the form $\langle A, S\rangle$ for certain kinds of properties (under the assumption that the analyzed programs are shallow). The intuitive meaning of a predicate of the form $\langle A, S\rangle$ is that all reference variables in the set A point to the same object (are aliased), and the object pointed to by these references is in one of the states in the set S. We also use a designated predicate *Error* that holds in a program-state if and only if the program-state contains an object in the error state *err*.

For the automaton of Fig. 1.3, the construction algorithm produces the predicates:

$$\langle\{f\}, \{open, closed, err\}\rangle, \langle\{f\}, \{closed, err\}\rangle, \langle\{f\}, \{err\}\rangle, \mathit{Error}$$

where $open$ is the initial state of a File component.

Using these predicates in an independent attribute analysis on the (shallow) example program of Fig. 1.5 results in the states shown in Fig. 1.8, thus proving that the property holds for the example program in polynomial time.

It is important to note that results in this chapter apply to programs that are shallow *with respect to the type being verified*, i.e., only objects of the verified type are required to be shallow and the rest of the heap may have an arbitrary depth.

```
[1]  while (...) {
[2]    f = new File();
[3]    if (chosen == null) {
[4]      if (?) {
[5]        chosen = f;
[6]      }
[7]    }
[8]    f.read();
[9]    f.close();
[10]   // do something
     }
```

Figure 1.9: An instrumented version of the example program instrumented to non-deterministically choose a single File component to be verified.

Using Separation

In Chapter 6, we further investigate the idea of guiding the abstraction by user-specification, and allow the user to provide a specification of how to decompose a verification problem into a number of independent subproblems that could be verified independently in an efficient manner. In this chapter, we also present a general framework of *heterogeneous abstractions* that allow different parts of the heap to be abstracted using different degrees of precision, at different points during the analysis. We show how to achieve more efficient verification by using the separation strategy (provided by the user) to transform (instrument) a verification problem instance (consisting of a safety property specification and an input program), and then utilize heterogeneous abstraction during the verification of the transformed verification problem.

For our example program (Fig. 1.5) and Property 1.1, separation amounts to the simple idea of verifying the property separately for each allocated file component. This could be viewed as verifying the property for a single *representative* file (the *chosen* one) in the instrumented code of Fig. 1.9. In this instrumented version of the program, each time a file is allocated, non-deterministic selection is applied to choose a single *chosen* file, if one has not been already selected (in this thesis we use '`?`' to denote a nondeterministic branch, as in line 4). As a result, in any execution of the instrumented program, at most one File component will be (nondeterministically) selected as the *chosen* one. Our verification procedure will only verify the correct usage of this *chosen* file component (referenced by `chosen`) in an execution. By exploring all possibilities of nondeterministic choice for the *chosen* file component, our verification method is guaranteed to represent all possible file components, and is therefore guaranteed to be sound. Using this form of separation allows us to successfully verify the desired property. The results of applying verification with separation to this example are shown in Fig. 1.10.

The example program and property used here are extremely simple. In Chapter 6, we handle a more general setting involving first-order safety properties.

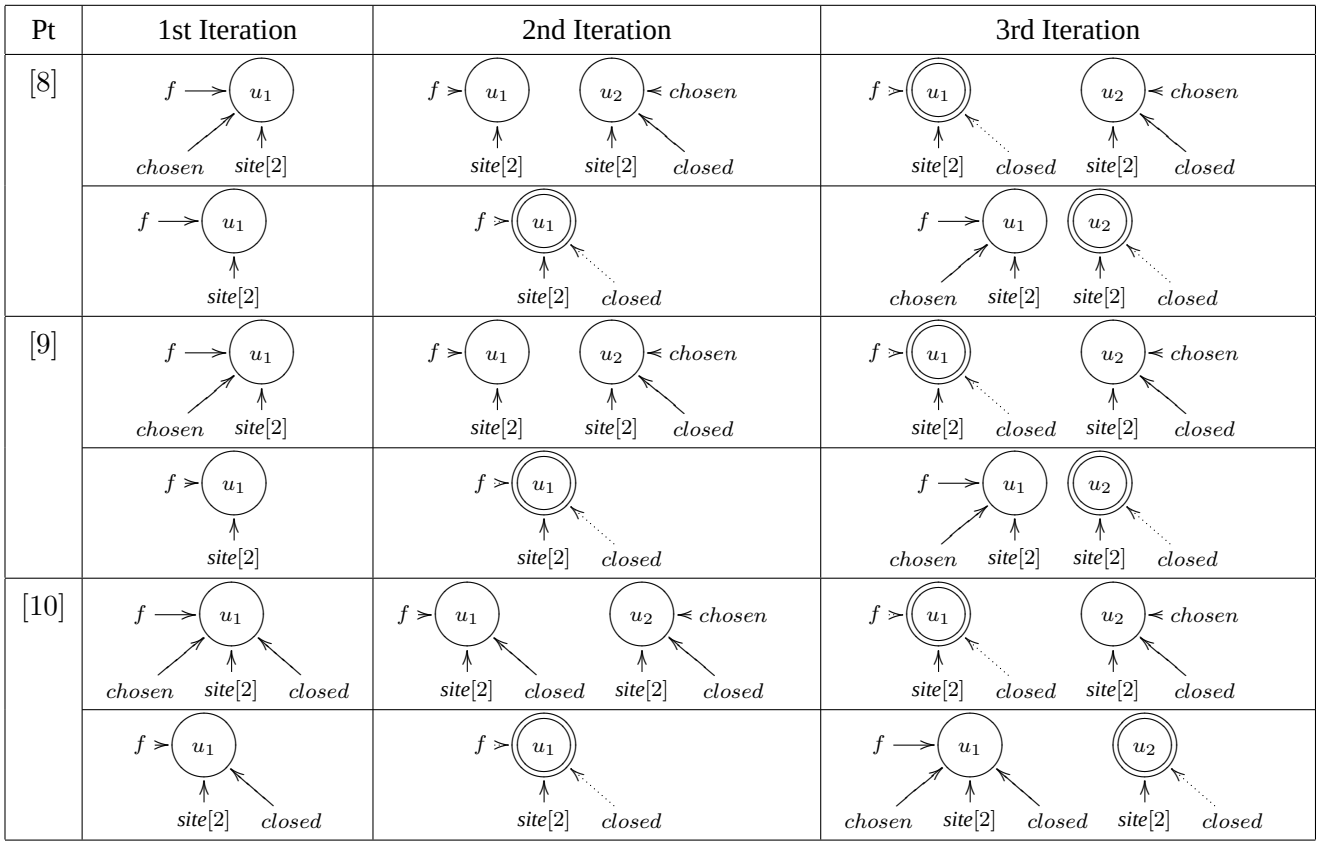


Figure 1.10: Analysis of the example program using simple separation.

1.3.4 Verifying Temporal Properties

In Chapter 4 and Chapter 5 we investigate two techniques for verifying general ETL specifications for heap-manipulating programs. For example, we can verify Property 1.2 for the program of Fig. 1.11. The technique of Chapter 5 performs verification by showing that the program does not satisfy the negation of the original specification (also referred to as the *violation property*). For Property 1.2, the violation property is:

$$\varphi = \Diamond \Box \, \forall v. \neg f(v) \vee closed(v)$$

In Chapter 5, we show how to derive the following set of predicates from the violation property, by taking the closure (all subformulae) of the ETLformula:

$$\{\langle \Diamond \Box \, \forall v. \neg f(v) \vee closed(v) \rangle, \langle \Box \, \forall v. \neg f(v) \vee closed(v) \rangle, \langle \forall v. \neg f(v) \vee closed(v) \rangle\}$$

These predicates correspond to future obligations that should be satisfied by the program's execution. We denote by $\langle \varphi \rangle$ the predicate recording the fact that φ should be satisfied by the future (or present) of program execution. Initially, we require that the future of the execution satisfies the violation property. As the analysis progresses, future obligations may be fulfilled, possibly leading to a repeatable state in which no further obligations exist, thus satisfying the violation property.

Fig. 1.12 shows part of the results produced in an attempt to verify Property 1.2 for the program of Fig. 1.11. Initially, the predicate $\langle \Diamond \Box \, \forall v. \neg f(v) \vee closed(v) \rangle$ holds, recording the fact that the future of the computation should satisfy the property $\Diamond \Box \, \forall v. \neg f(v) \vee closed(v)$. When the analysis reaches line 5, the single file component becomes closed, satisfying the local property $\forall v. \neg f(v) \vee closed(v)$, and thus possibly starting a continuous sequence for which $\Box \, \forall v. \neg f(v) \vee closed(v)$ holds. As a result, our analysis takes two possibilities into account (producing two possible configurations at this point): (i) the future of the computation from this point on satisfies $\Box \, \forall v. \neg f(v) \vee closed(v)$; (ii) the property has not stabilized yet, and the future of the computation should satisfy the initial property $\Diamond \Box \, \forall v. \neg f(v) \vee closed(v)$. However, when the configuration recording possibility (i) above reaches line 3, the property $\Box \, \forall v. \neg f(v) \vee closed(v)$ no longer holds, and this configuration is not propagated any further. When the analysis terminates, it does so without finding a configuration that satisfies the violation property, thus showing that Property 1.2 holds for the example program.

The example program and property used here are very simple. In particular, Property 1.2 does not relate individuals (objects) across configurations and is essentially a propositional property over propositions extracted from a first-order configuration. In Chapter 4 we refer to such specifications as *temporally separable* specifications, and handle more general properties that may relate individuals of different configurations.

The technique presented in Chapter 4 operates directly on abstract representation of traces and provides a conceptual model for the verification of heap-manipulating programs. In this technique, a possibly infinite set of infinite traces is finitely represented by an abstract trace. ETL properties are translated

```
[1]  while (true) {
[2]    f = new File();
[3]    f.read();
[4]    f.close();
[5]    // do something
     }
```

Figure 1.11: A simple example program reading from a component referenced by `f` infinitely often.

into FO^{TC} formulae that are evaluated directly over a first-order representation of abstract traces. For brevity, we do not demonstrate the application of this technique in this overview section.

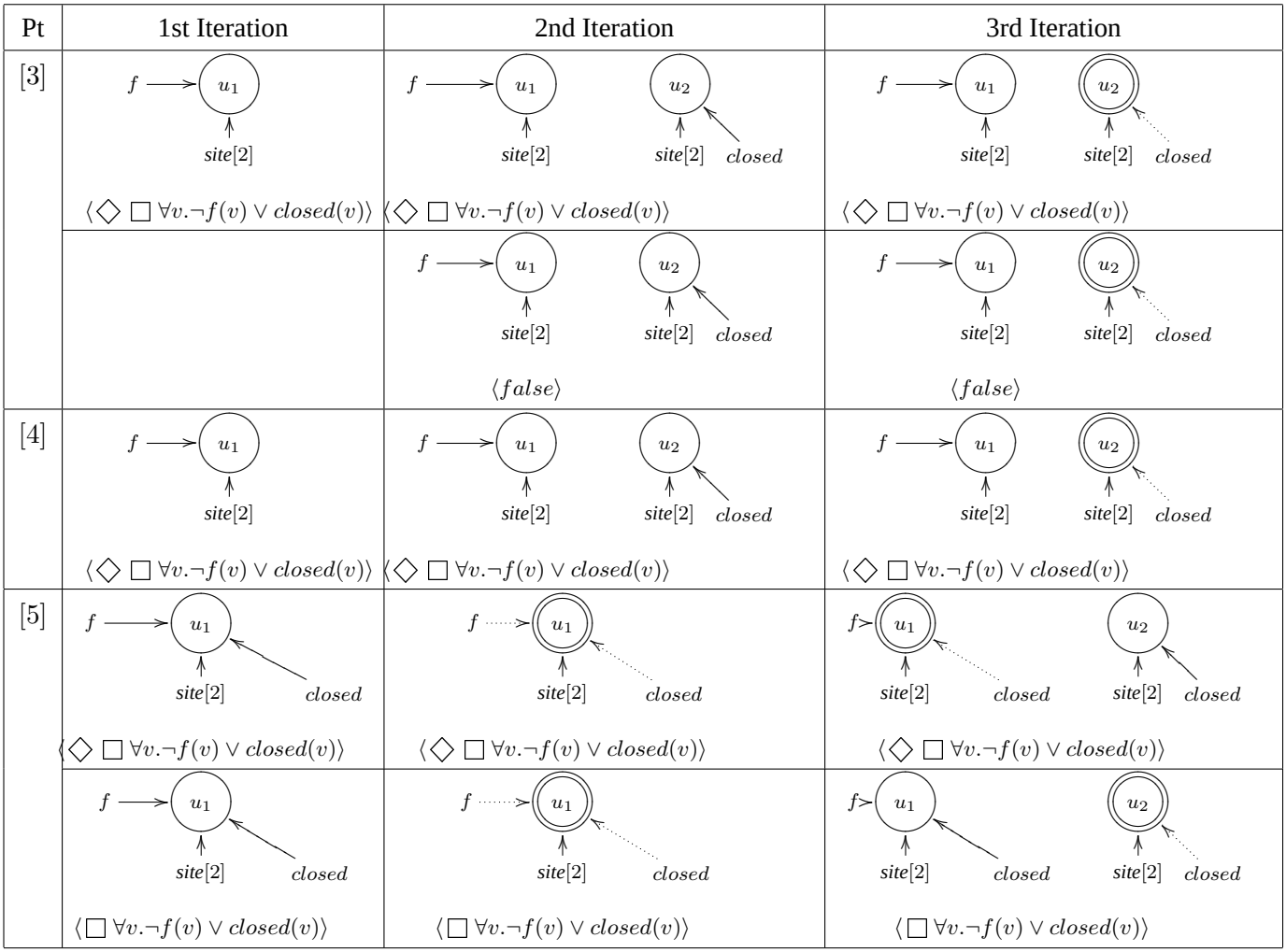


Figure 1.12: Verifying property 1.2 for the program of Fig. 1.11

Chapter 2

Verifying Safety Properties of Concurrent Java Programs Using 3-Valued Logic

We provide a parametric framework for verifying safety properties of concurrent Java programs. The framework combines thread-scheduling information with information about the shape of the heap. This leads to error-detection algorithms that are more precise than existing techniques. The framework also provides the most precise shape-analysis algorithm for concurrent programs. In contrast to existing verification techniques, we do not put a bound on the number of allocated objects. The framework even produces interesting results when analyzing Java programs with an unbounded number of threads. The framework is successfully applied to verify the following properties of a concurrent program:

- Concurrent manipulation of linked-list based ADT preserves the ADT datatype invariant. When applied to concurrent queue implementations, this allows proving the correctness of the queue algorithms (e.g., two-lock queue).
- The program does not perform inconsistent updates due to interference.
- The program does not reach a deadlock.
- The program does not produce runtime errors due to illegal thread interactions.

We also find bugs in erroneous versions of such implementations.
A prototype of our framework has been implemented and applied to interesting example programs.

"I have now in my hands," my companion said, confidently,
"all the threads which have formed such a tangle..."
–Sir Arthur Conan Doyle, *A Study in Scarlet*.

2.1 Introduction

Java provides low-level concurrency-control constructs that enable the programmer to create complicated and powerful synchronization schemes. The Java language provides no means for compile-time checking and almost no means for runtime checking of the correctness of concurrent behavior. This makes concurrent programming in Java quite error-prone (e.g., [108]).

The theme of this chapter is to develop compile-time techniques for verifying safety properties by detecting program configurations that may violate desired properties. This is a different task than dynamic anomaly-detection techniques, which operate on a given input (and thus can only show the presence of errors, not their absence).

2.1.1 Main Results and Related Work

In this chapter, we present a framework for verifying safety properties of concurrent Java programs. This framework handles dynamic allocation of objects and references to objects. This allows us to analyze programs that dynamically allocate thread objects, and even programs that create an unbounded number of threads. Dynamic allocation of threads is common when implementing services in threads (e.g., [63], ch. 6). For these programs, we can verify properties such as the absence of interference. Handling dynamically allocated objects also allows us to model concurrent programs that manipulate linked-lists in the most precise known way.

A Parametric Framework for Verifying Safety Properties

We provide a parametric framework for verifying safety properties of concurrent Java programs. We use different instances of this framework (see Section 2.1.1) to obtain static-analysis algorithms that have the ability to verify different safety properties.

The semantics of Java can be described using a structural operational semantics (e.g., [59]) in terms of *configurations* (or states). In our framework, the operational semantics of Java statements (and conditions) is specified using a meta-language based on first-order logic with transitive-closure. The same meta-language is also used to check that a safety property holds in a given configuration. Our framework then computes a safe approximation of the (usually infinite) set of *reachable configurations*, i.e., configurations that can arise during program execution. This can be formulated within the theory of abstract interpretation [24]. The main idea is to conservatively represent many configurations using a single *abstract configuration*. The effect of every statement (and condition) on an abstract configuration is then conservatively computed, yielding another abstract configuration. Also, the framework conservatively verifies that all the "reachable abstract configurations" satisfy the desired safety property. Thus, we may falsely report that a safety property may be violated (*false alarm*) but can never miss a violation.

Our framework can be viewed as on-the-fly model checking [17] for verifying safety properties of programs. On-the-fly model checking does not require the construction of a global state graph as a prerequisite for property verification. In order to handle dynamic creation and references to objects, we use first-order logical structures to represent configurations of the program. A *state-space exploration* algorithm (see Fig. 2.4) is used to generate the configurations *reachable* from an initial set of configurations. The effect of every program statement is modeled by *actions* specified using *first-order logical formulae*. Our abstract configurations are bounded representations of logical structures. A (concrete) configuration is automatically abstracted into an abstract configuration.

Many approaches were proposed to handle verification of unbounded data structures. Traditional approaches consist of manually abstracting the data-structure into a simple finite state machine representing the states of the data-structure that are relevant to the verification problem (e.g., [101, 103]). Other, more recent approaches, use a combination of theorem-proving and model checking techniques to automatically construct such abstractions [1, 8, 9].

Our framework should be contrasted with traditional model checking algorithms in which a bounded representation is guaranteed by using *propositional formulae* for actions. Moreover, most model checking techniques perform an abstraction when the model is extracted, and apply actions with a fixed number of propositional variables ([16]). This could be trivially encoded in our framework by using only nullary predicates (and thus the number of individuals in a logical structure is immaterial). In fact, our framework allows more general (and natural) modeling of programs by using unary and binary predicates. This is crucial in order to handle dynamically allocated objects and references to objects where the "name" of the object is unknown at compile-time. Even the technique of [38] (formulated for processes rather than threads) relies on explicit process names, and thus cannot handle dynamic allocation of processes.

ESP [27] and SLAM [72] use a preceding pointer-analysis phase and use the results of this phase to perform finite-state verification of sequential programs. Separating verification from pointer-analysis may generally lead to imprecise results (see Section 1.3.2). In contrast, our framework handles concurrent programs, and applies integrated verification and pointer analysis which is more precise.

Bandera [19] is a framework for translating Java programs to a program model acceptable by existing model checking tools. During translation, the model is reduced using slicing and other program analyses.

[90] presents a new modular and customizable model checking framework. Similarly to ESP and SLAM, this framework assumes that pointer-analysis is applied as a preprocessing phase, prior to verification.

JavaPathFinder [52] and Java2Spin [31] translate Java source code to PROMELA representation. The SPIN model-checker [55] is then used to verify properties of the PROMELA program. Both these tools put a bound on the number of allocated objects since it is imposed by SPIN. A variant of SPIN

named dSPIN [32] supports dynamic allocation of objects. However, since it uses no abstraction, it can only handle bounded data-structures and a bounded number of threads. [14] presents a method for the verification of parametric families of systems. A network grammar is used to construct a process invariant that simulates all systems in the family. However, it cannot handle dynamic allocation of objects.

Das, Dill, and Park [28] have used predicate abstraction to verify the properties of a cache coherence algorithm and a concurrent garbage-collection algorithm. The garbage collection algorithm was verified in the presence of a single mutator thread executing concurrently with the collector.

Saidi [92] presents new abstraction predicates but does not have the notion of summary nodes. Thus, it cannot handle programs with an unbounded number of allocated objects. Moreover, our framework presents a model checking algorithm that recognizes abstraction as suggested there.

In our framework, rather than having separate model-extraction and model checking phases, we follow the abstract-interpretation approach [24] and cast our analysis in a syntax-directed manner.

Technically speaking, our framework is a generalization of [91] in the following aspects: (i) Program configurations are used to model the global state of the program instead of modeling only the relationships between heap-allocated objects. This allows us to combine thread scheduling information with information about the shape of the heap. (ii) Program control-flow is not separately represented, but instead the program location of each thread is maintained in the configuration which allows us to handle an unbounded number of threads in a natural way. This is naturally coded in first-order logic as a property of a thread (in contrast to model checking in which it is externally coded). Furthermore, it does not require control-flow information to be computed in a separate earlier phase. This is an advantage because the imprecision in control-flow computation could lead to imprecise results. (iii) We use the standard interleaving model of concurrency. A slightly different generalization is used in [79], which even allows the program to modify itself to support the semantics of Mobile Ambients [12].

The FLAVERS system [76] uses trace flow graphs with feasibility constraints, represented as finite-state automata, to model the semantics of concurrent Java programs. An important difference between our framework and FLAVERS is that our framework has the ability to model the dynamic creation of objects and threads. Moreover, since every finite automaton can be coded in our framework, it generalizes FLAVERS. However, the cost of doing that in our current implementation may be higher.

In [98], a framework for model checking distributed Java programs is presented. This framework uses partial-order methods to reduce the size of the explored state-space. However, it uses no abstraction and thus can only handle bounded data structures and a bounded number of threads. We intend to use similar partial-order methods in future versions of our framework.

In [18], shape analysis of concurrent programs is used to reduce finite-state models of concurrent Java programs. In this analysis, the number of threads is bounded. The algorithm presented is based on [13], which uses a single *shape graph* for each program location, and uses an abstraction which leads to

overly imprecise results (e.g., in programs that traverse data structures based on allocation sites).

In [104], shape analysis of concurrent programs is used for eliminating synchronization. As in [18], the algorithm presented is an extension of [13] and suffers from the same imprecision. It should be noted that despite the different goals of our work, it is significantly more precise. In particular, it always performs strong updates.

In [3], static analysis is used to identify opportunities to eliminate unnecessary synchronization. That work assumes a static control-flow-graph, and ignores thread-scheduling mechanisms.

Applications

We have used our framework to verify the properties listed below.

Interference: Two threads are said to *interfere* when they may both access a shared object simultaneously, and at least one of them is performing an update of the shared object. We use our framework to locate read-write and write-write interference between threads (see [78]). Here, we benefit from the fact that the analysis keeps track of both scheduling information and information about the shape of the heap. For example, in a two-lock-queue (see [71], also shown in the appendix) we are able to show that write-write interference is not possible since writing is never performed on the same object.

Deadlock: Our framework has been used to verify the absence of a few types of deadlocks: (i) total deadlocks in which all threads are blocked. (ii) nested monitors deadlocks, which are very common in Java ([108]) (iii) partial deadlocks created by threads cyclically waiting for one another.

We are also able to verify that a program complies with a resource-ordering policy, and thus cannot produce a deadlock (see [63], ch. 8).

Shared ADT: Our framework has been used to verify that a shared ADT, based on a linked-list, preserves ADT properties under concurrent manipulation. Here, the strength of our technique is obvious, since precise information about the structure of a scheduling queue can be used to precisely reason about thread scheduling. In particular, our framework has been applied to verify the concurrent queue algorithms presented by Michael and Scott in [71] which are in part implemented in the `java.util.concurrent` package of JDK1.5. (a preliminary version of this case study appeared in [120]).

Our framework has also been applied to prove the correctness of the apprentice challenge, originally presented by J. Moore as a challenge for Java verification [73].

For example, Fig. 2.1(a) shows a concurrent program using a queue. The implementation of the queue is given in Fig. 2.1(b) and Fig. 2.2. This program is used as a running example throughout this chapter. Our technique is able to show that the properties of the queue are correctly maintained by this program without any *false alarms*. Moreover, since the analysis is conservative, it is guaranteed to report errors when analyzing an ill-synchronized version of the same queue (not shown here).

Illegal Thread Interactions: The Java semantics allows the programmer to introduce thread interactions that are illegal and result in an exception during program execution (this is the only runtime checking applied by Java for correctness of concurrent behavior). For example — starting a thread more than once will result with an `IllegalThreadStateException` being thrown. Our framework has been used to detect such illegal interactions.

Prototype Implementation

We have implemented a prototype of our framework called 3VMC [113]. In Section 2.6, we report experimental results of applying this prototype to several small but interesting programs. We then show a detailed case study of applying our framework to verify the correctness of concurrent queue algorithms.

Currently, we do not perform interprocedural analysis and assume that procedures are inlined. Support for (recursive) procedures can be added by extending the approach described by [88].

The main disadvantage of our current implementation is that no optimizations are used, and thus only small programs can be handled. However, we are encouraged by the precision of our results and the simplicity of the implementation.

Outline of this Chapter

In Section 2.2, we give a brief overview of Java's concurrency model. Section 2.3 defines our formal model which uses logical structures to represent program configurations. Section 2.4 shows how multiple program configurations can be conservatively represented using a 3-valued logical structure. In Section 2.5, we show how our method can be used to detect several common concurrency errors. In Section 2.6, we describe the prototype implementation and results we have obtained using it to analyze a few small but interesting programs. Application of the framework to more realistic examples is described in Chapter 7.

2.2 Java Concurrency Model

We now give a short description of the Java concurrency-primitives used in this thesis. The reader is referred to [47, 63, 67] for more details.

Java contains a few basic constructs and classes specifically designed to support concurrent programming:

- The class `java.lang.Thread`, used to initiate and control new activities.
- The `synchronized` keyword, used to implement mutual exclusion.

```
  class Producer implements Runnable {
    protected Queue q;
    ...
    public void run() {
      ...
      q.put(val1);
    }
  }
  class Consumer implements Runnable {
    protected Queue q;
    ...
    public void run() {
      ...
      val2 = q.take();
    }
  }
  class Approver implements Runnable {
    protected Queue q;
    ...
    public void run() {
         q.approveHead();
    }
}
  class Main {
    public static void main(String[] args) {
lm1 Queue q = new Queue();
lm2 Thread prd = new Thread(new Producer(q));
lm3 Thread cns = new Thread(new Consumer(q));
lm4 for(int i = 0; i < 3; i++) {
lm5       new Thread(new Approver(q)).start();
    }
lm6 prd.start();
lm7 cns.start();
    }
  }
```

(a)

```
 // Queue.java
  class Queue {
    private QueueItem head;
    private QueueItem tail;
    ...
    public void put(int value) {
lp1       QueueItem x_i = new QueueItem(value);
lp2       synchronized(this) {
lp3           if (tail == null) {
lp4              tail = x_i;
lp5              head = x_i;
              } else {
lp6              tail.next = x_i;
lp7              tail = x_i;
              }
lp8       }
lp9   }
    public QueueItem take() {
lt1       synchronized(this) {
              QueueItem x_d = null;
lt2           if (head != null) {
lt3              newHead = head.next;
lt4              x_d = head;
lt5              x_d.next = null;
lt6              head = newHead;
lt7              if (newHead == null) {
lt8                 tail = null;
                 }
              }
lt9       }
lt10  return x_d;
    }
    public void approveHead() {
la1       synchronized(this) {
la2           if (head != null)
la3              head.approve();
la4       }
    }
  }
```

(b)

Figure 2.1: (a) a simple program that uses a queue, (b) simplified Java source code for a queue implementation.

```
// QueueItem.java
 class QueueItem {
   private QueueItem next;
   private int value;
   private boolean isApproved;
   ...
   public void approve() {
     ...
   }
 }
```

Figure 2.2: Simplified Java source code for a QueueItem implementation.

- The methods `wait`, `notify`, and `notifyAll` defined in `java.lang.Object`, used to coordinate activities across threads.

The constructor for the `Thread` class takes an object implementing the `Runnable` interface as a parameter. The `Runnable` interface requires that the object implement the `run()` method. Threads may be also created directly without a `Runnable` object, by inheriting from the `Thread` class, and overriding the `run()` method. However, in this thesis we prefer to always use Thread construction with a `Runnable` object.

A thread is *created* by executing a `new Thread()` allocation statement. A thread is *started* by invoking the `start()` method and starts executing the `run()` method of the object implementing the `Runnable` interface.

Initially, a program starts with executing the `main()` method by the main thread. Java assumes that threads are scheduled arbitrarily.

The program shown in Fig. 2.1(a) contains 3 classes implementing the `Runnable` interface: a `Producer` class, which puts items into a shared queue; a balking `Consumer` class, which takes items from a shared queue and does not wait for an item if the queue is empty; and an `Approver` class, which performs some computation on a queue element to approve it. The program starts by executing the `main()` method, which creates a shared queue, a Producer thread, a Consumer thread, and 3 Approver threads. Threads in the example are started at labels lm_5, lm_6, and lm_7.

Each Java object has a unique implicit lock associated with it. In addition, each object has an associated block-set and wait-set for managing threads that are blocked on the object's lock or waiting on the object's lock. When a `synchronized(expr)` statement is executed by a thread t, the object expression $expr$ is evaluated, and the resulting object's lock is checked for availability. If the lock has not been *acquired* by any other thread, t *successfully acquires* it. If the lock has already been acquired by another thread t', the thread t becomes *blocked* and is inserted into the lock's block-set. A thread

may acquire more than one lock, and may acquire a lock more than once. When a thread leaves the $synchronized$ block, it *unlocks* the lock associated with it. When a lock has been acquired more than once (by the same thread), it is released only when a matching number of *unlock* operations is performed.

In the example shown in Fig. 2.1, we guarantee that the queue operations are atomic by putting critical code into a `synchronized(this)` block.

A thread t may become *waiting* on a lock l by invoking a call to `o.wait()` on l's object (o); a call to `o.wait()` puts t in l's wait-set.

When t becomes waiting on l, it releases the lock l, but does not release any other locks it acquired. A *waiting* thread t can be only released by another thread invoking `o.notify()`, `o.notifyAll()` for the lock l or `interrupt()` on the thread t.

Invoking `notify()` on an object removes an arbitrary thread from the object's wait-set, and makes it available for scheduling. Invoking `notifyAll()` on an object, removes all threads from the wait-set, and makes them available for scheduling.

A thread t should only invoke `wait()`, `notify()` and `notifyAll()` when it is holding the object's lock, otherwise an exception is thrown.

A thread t_1 may wait for another thread t_2 to complete execution and *join* it, by invoking a call to $t_2.join()$. If t_2 is not yet started or t_2 is already dead, the call for $t_2.join()$ is ignored.

Java uses a variant of no-priority non-blocking monitors [11]. In no-priority monitors a notified thread has no priority over blocked threads, or over a thread just reaching the monitor entrance. Notified threads, blocked threads, and entering threads have the same priority when competing to acquire a lock. Therefore, a notified thread does not resume execution immediately, but is moved to the block-set, and competes to re-acquire the lock.

For the sake of simplicity and readability we make the following simplifying assumptions:

- We assume the identity of the lock for `synchronized(exp)`, and the target object of scheduling-related methods, is given as a single reference variable rather than a general reference expression as supported by the Java language. If the program uses a general expression, we normalize the program by adding a temporary variable.

- Similarly, we assume the target object of scheduling-related methods (`notify()`, `notifyAll()`, `wait()` etc.) is given as a single reference variable.

- We assume that the memory-model provides sequential consistency. This assumption abstracts away the actual details of the memory model and is common to all Java verification frameworks. While our framework is expressive enough for expressing the lower-level semantics involving the actual memory-model, the behavior of that model under abstraction remains an issue for further research.

- For the sake of clarity, we do not present here the semantics for multiple acquisitions of a lock by the same thread.

- We may handle additional Java features such as exceptions and dynamic binding in a conservative manner.

2.3 A Program Model

In this section we lay the ground for our analysis of Java programs. In Section 2.3.1 we use logical structures to represent the global state of a multithreaded program. Section 2.3.2 uses logical formulae as meta-language to extract interesting properties of a configuration such as mutual exclusion. Then, in Section 2.3.3, we define a structural small step operational-semantics which manipulates configurations using logical formulae. Finally, in Section 2.3.4, we describe the safety properties that are verified in this chapter.

2.3.1 Representing Program Configurations via Logical Structures

First-order logical structures provide a natural formalism for representing the global state of a heap-manipulating program — individuals of the first-order structure correspond to heap-allocated objects, properties of objects are represented using unary predicates, and relationships between objects are represented using binary predicates. It is also possible to use first-order logical structures to model non heap-allocated objects (such as integer values), as well as enforce a typing mechanism on objects by using a unary predicate $is\_T(v)$ to denote objects of type T.

A *program configuration* encodes a global state of a program which consists of (i) a global store, (ii) the program-location of every thread, and (iii) the status of locks and threads, e.g., if a thread is waiting for a lock. Technically, first-order logic with transitive-closure is used in this chapter to express configurations and their properties in a parametric way. Formally, we assume that there is a set of predicate symbols P for every analyzed program each with a fixed arity. Table 2.1 contains the predicates used to analyze our example programs.

- The unary predicate $\epsilon(v)$ holds for objects that exist in the current configuration.

- The binary predicate $eq(v_1, v_2)$ holds for objects that are equal.

- A unary predicate $is\_T(v)$ is used to denote the objects of type T. In particular, the unary predicate $is\_thread(t)$ denotes objects that are threads, i.e., instances of the `java.lang.Thread` or its subclasses.

Predicates	**Intended Meaning**
$\epsilon(v)$	v exists in the configuration
$eq(v_1, v_2)$	v_1 equals to v_2
$is\_T(v)$	v is an object of type T
$zero(v)$	the individual v represents integer value zero
$succ(v_1, v_2)$	v_2 is the successor value of v_1
$\{at[lab](t) : lab \in Labels\}$	thread t is at label lab
$\{rv[fld](v_1, v_2) : fld \in RFields\}$	field fld of the object v_1 points to the object v_2
$\{iv[fld](v_1, v_2) : fld \in IFields\}$	field fld of the object v_1 has the value v_2
$held\_by(l, t)$	the lock l is held by the thread t
$blocked(t, l)$	the thread t is blocked on the lock l
$waiting(t, l)$	the thread t is waiting on the lock l

Table 2.1: Predicates for partial Java semantics.

- To model integer values, we introduce objects of type unsigned-integer, where the unary predicate $zero(v)$ is used to record the integer with the value zero, and the binary predicate $succ(v_1, v_2)$ to record the successor relationship between integers.

- For every potential program-location (program label) lab of a thread t, there is a unary predicate $at[lab](t)$ which is true when t is at lab.

- For every class field and function parameter `fld`, a binary predicate $rv[fld](v_1, v_2)$ records the fact that the `fld` field of the object v_1 points to the object v_2.

- For every integer valued field `ifld`, a binary predicate $iv[fld](v_1, v_2)$ that represents the integer value of a field by relating an object v_1 to an individual representing an integer value v_2.

- The predicates $held\_by(l, t)$, $blocked(t, l)$ and $waiting(t, l)$ model possible relationships between locks and threads. $held\_by(l, t)$ is true when the lock l has been acquired by the thread t, via a successful `synchronized` statement. $blocked(t, l)$ is true when the thread t is blocked on the lock l, as a result of an unsuccessful `synchronized` statement. $waiting(t, l)$ is true when the thread t is waiting for the lock l as a result of invoking a `wait()` call.

Note that predicates in Table 2.1 are actually written in a generic way and can be applied to analyze different Java programs by modifying the set of labels and fields.

A *program configuration* is a 2-valued logical structure $C^\natural = \langle U^\natural, \iota^\natural \rangle$ where:

- $U^\natural$ is the infinite universe of the 2-valued structure. Each individual in $U^\natural$ represents an allocated

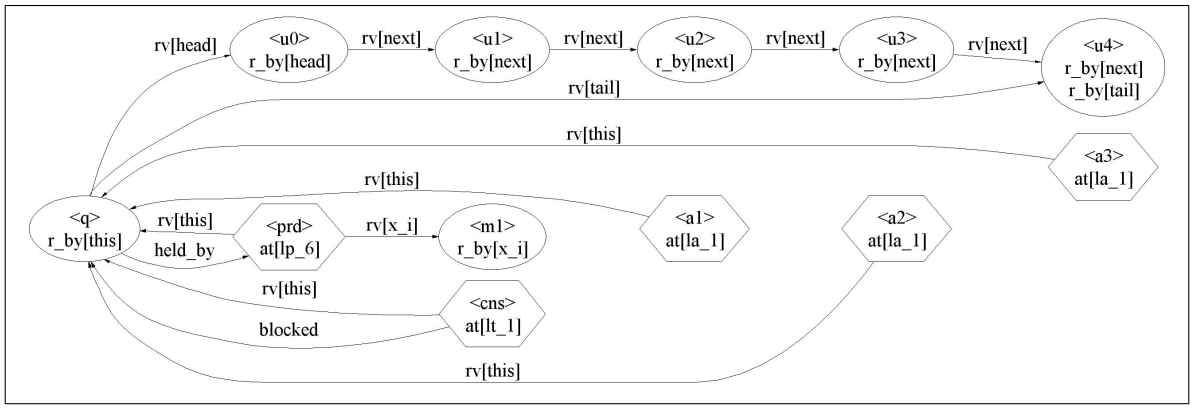


Figure 2.3: A concrete configuration $C^{\natural}_{2.3}$.

heap object (some of which may represent the threads of the program, and the configuration also contains an infinite number of individuals representing the unsigned integers).

- $\iota^{\natural}$ is the interpretation function mapping predicates to their truth-value in the structure, i.e., for every predicate $p \in P$ of arity k, $\iota^{\natural}(p) \colon {U^{\natural}}^k \to \{0, 1\}$.

Usually, not all logical structures represent valid program configurations, therefore TVLA/3VMC allows the programmer to introduce integrity constraints specified as FO^{TC} (first order-logic with transitive closure) formulae [91]. The integrity constraints for integers are simply the Peano axioms encoded using FO formulae.

In this thesis, program configurations are depicted as directed graphs. Each existing individual of the universe (one for which ϵ holds) is displayed as a node — objects of type thread are presented as hexagonal nodes, other objects as round nodes. A unary predicate p which holds for an individual (node) u is drawn inside the node u. Since only objects for which ϵ holds are shown, we do not draw this predicate. In some of the figures, we use node names written inside angle brackets. Node names are only used for ease of presentation and do not affect the analysis. A true binary predicate $p(u_1, u_2)$ is drawn as directed edge from u_1 to u_2 labeled with the predicate symbol. For brevity, the predicate $eq(v_1, v_2)$ is not shown, and the integer nodes are omitted when possible. We use a *natural* sign ($\natural$) to denote entities of the concrete domain (e.g., $C^{\natural}$ denotes a concrete configuration C).

Example 2.3.1 *The configuration $C^{\natural}_{2.3}$ shown in Fig. 2.3 corresponds to a global state of the example program with 5 threads: a single* producer *thread (labeled prd) which acquired the queue's lock, a single* consumer *thread (labeled cns) which is blocked on the queue's lock, and 3* approving *threads ($a1, a2, a3$) which haven't performed any action yet. The role of the predicate $r\_by[fld](o)$ will be explained in future sections. For clarity of presentation, we omit the* `Runnable` *objects and present only thread objects.*

All threads in the example use a single shared queue containing 5 items $\{u0, \ldots, u4\}$. The binary predicate $rv[next](o_1, o_2)$ records for each object o_1 the target object referenced by its `next` *field.*

Note that the number of elements that actually exist in a universe is not bounded since the analyzed program may allocate new non-thread individuals, new thread individuals, or both. We do not place a bound on the number of allocated objects.

2.3.2 Extracting Properties of Configurations using Logical Formulae

Properties of a configuration can be extracted by evaluating a first-order logical formulae with transitive closure (FO^{TC}) over configurations. The (standard) syntax and semantics of FO^{TC} are given in Appendix A.

In this thesis, we are mostly interested in properties that hold for objects that actually exist in a configuration. We therefore define the following notion of relativization for a formula in negation normal form (where negations only appear over predicates).

Definition 2.3.2 *Given an FO^{TC} formula φ in negation normal form (NNF), we define the* ***relativization*** *$(\varphi)^\epsilon$ of φ as follows:*

$$
\begin{aligned}
&(1)^\epsilon = 1\\
&(0)^\epsilon = 0\\
&(p(v_1, \ldots, v_k))^\epsilon = \textstyle\bigwedge_{1 \le i \le k} \epsilon(v_i) \wedge p(v_1, \ldots, v_k)\\
&(\neg p(v_1, \ldots, v_k))^\epsilon = \textstyle\bigwedge_{1 \le i \le k} \epsilon(v_i) \wedge \neg p(v_1, \ldots, v_k)\\
&(\varphi \wedge \psi)^\epsilon = (\varphi)^\epsilon \wedge (\psi)^\epsilon\\
&(\varphi \vee \psi)^\epsilon = (\varphi)^\epsilon \vee (\psi)^\epsilon\\
&(\exists v.\varphi(v))^\epsilon = \exists v.\epsilon(v) \wedge (\varphi(v))^\epsilon\\
&(\forall v.\varphi(v))^\epsilon = \forall v.\epsilon(v) \implies (\varphi(v))^\epsilon\\
&((TC\ v1, v2\colon \varphi)(v3, v4))^\epsilon = (TC\ v1, v2\colon (\varphi)^\epsilon \wedge \epsilon(v1) \wedge \epsilon(v2))(v3, v4)\\
&((NTC\ v1, v2\colon \varphi)(v3, v4))^\epsilon = (NTC\ v1, v2\colon (\varphi)^\epsilon \vee \neg(\epsilon(v1) \wedge \epsilon(v2)))(v3, v4)
\end{aligned}
$$

For example, the following formula describes the fact that a lock pointed-to by the `this` field of a thread, has been acquired by the thread, and is now being held by the thread.

$$\exists t, l.is\_thread(t) \wedge rv[this](t, l) \wedge held\_by(l, t)$$

the relativization of this formula is

$$\exists t, l.\epsilon(t) \wedge \epsilon(l) \wedge is\_thread(t) \wedge rv[this](t, l) \wedge held\_by(l, t)$$

which means that this formula will only evaluate to true in a configuration where both the thread t and the lock l actually exist.

In the rest of this chapter, unless stated otherwise, we assume that formulae are implicitly normalized to NNF and relativized before evaluation

For ease of notation, we use the shorthand $\forall v\colon type.\varphi \triangleq \forall v.is\_type(v) \rightarrow \varphi$. Which allows us to write the above formula in a more readable form as:

$$\exists t\colon thread \exists l.rv[this](t,l) \wedge held\_by(l,t)$$

Our experience indicates that it is quite natural to express configuration properties using first-order logics.

Transitive closure is useful in the running example for expressing reachability. For example, the fact that an element u_1 in the queue q is reachable from `head`, we write the formula:

$$\exists u.rv[head](q,u) \wedge rv[next]^*(u,u_1)$$

Note that the program-location of each thread can be used in a formula by using the appropriate label. For example, consider a label l_{crit} which corresponds to a critical section. We formalize the mutual exclusion requirement using the following formula:

$$\forall t_1,t_2\colon thread.(t_1 \neq t_2) \rightarrow \neg(at[l_{crit}](t_1) \wedge at[l_{crit}](t_2))$$

2.3.3 A Structural Operational Semantics of Configurations

Fig. 2.4 shows a state-space exploration. For each configuration C such that C is not already a *member* of the *state-space*, we explore every configuration C' that can be produced by applying some action to the current configuration C.

Every resulting configuration C', is added to the *state-space* using set union. The membership operator used is set-membership, we will later use a generalized membership operator. In the case of set membership, this algorithm is essentially the classic state-space exploration used in model checking [17]. However, in contrast to model checking, there is no bound on the number of objects, and therefore the state-space explored by this algorithm is not guaranteed to be finite. A possible solution for this problem is given in Section 2.4.

Informally, an *action* is characterized by the following kinds of information:

- The *precondition* under which the action is enabled, expressed as logical formula. This formula may also include a designated free variable t_s to denote the "scheduled" thread on which the action is performed. Our operational semantics is non-deterministic in the sense that many actions can be enabled simultaneously and one of them is chosen for execution. In particular, it selects the scheduled thread by an assignment to t_s. This implements the interleaving model of concurrency.

```
initialize(C_0) {
  WorkSet = C_0
}

explore() {
  while WorkSet is not empty {
    select and remove C from WorkSet
    if not member(C, stateSpace) {
      verify(C)
      stateSpace' = stateSpace ∪ {C}
      for each action ac
        for each C' such that C ⇒_ac C'
          WorkSet = WorkSet ∪ {C'}
    }
  }
}
```

Figure 2.4: State space exploration.

- Enabled actions create a new configuration where the interpretations of every predicate p of arity k is determined by evaluating a formula $\varphi_p(v_1, v_2, \ldots, v_k)$ which may use $v_1, v_2, \ldots, v_k$ and t_s as well as all other predicates in P.

Table 2.2 defines the semantics of concurrency statements used in the running example. The table lists a precondition and update formulae for each action. The value of a predicate $p(v_1, v_2, \ldots, v_k)$ after the update is given by a formula $\varphi_{p(v_1,v_2,\ldots,v_k)}$. Predicates not given an update formulae are assumed to remain unchanged by the action. The set of actions is partitioned to blocking and non-blocking actions. Blocking actions do not affect the program-location. Non blocking actions advance to the next program-location by updating the $at[lab](t_s)$ predicates for the thread.

A Java statement may be modeled by several alternative actions corresponding to the different behaviors of the statement.

When a precondition is enabled, it determines a thread (denoted by t_s) that executes the action, and an action to be taken. A Java statement may be modeled by several alternative actions corresponding to the different behaviors of the statement.

The actions $lock(v)$ and $blockLock(v)$ correspond to the two possible behaviors on entry to a `synchronized(v)` block: $lock(v)$ is enabled when there exists no thread (other than the current thread) that is holding the lock referenced by v, $blockLock(v)$ is enabled when such a thread exists. The action $unlock(v)$ corresponds to the release of the lock upon exit of the `synchronized(v)` block. The action $wait(v)$ corresponds to invocation of `v.wait()`. The actions $notify(v)$ and

Action	**Precondition**	**Predicate-update**
$lock(v)$	$\neg\exists t \neq t_s.rv[v](t_s, l)$ $\wedge\, held\_by(l, t)$	$\varphi_{held\_by(l_1,t_1)} = held\_by(l_1, t_1) \vee (t_1 = t_s \wedge l_1 = l)$ $\varphi_{blocked(t_1,l_1)} = blocked(t_1, l_1) \wedge ((t_1 \neq t_s) \vee (l_1 \neq l))$
$unlock(v)$	$rv[v](t_s, l)$	$\varphi_{held\_by(l_1,t_1)} = held\_by(l_1, t_1) \wedge (t_1 \neq t_s \vee l_1 \neq l)$
$wait(v)$	$rv[v](t_s, l)$	$\varphi_{held\_by(l_1,t_1)} = held\_by(l_1, t_1) \wedge (t_1 \neq t_s \vee l_1 \neq l)$ $\varphi_{waiting(t_1,l_1)} = waiting(t_1, l_1) \vee (t_1 = t_s \wedge l_1 = l)$
$notify(v)$	$rv[v](t_s, l)$ $\wedge\, waiting(t_w, l)$	$\varphi_{waiting(t_1,l_1)} = waiting(t_1, l_1) \wedge (t_1 \neq t_w \vee l_1 \neq l)$ $\varphi_{blocked(t_1,l_1)} = blocked(t_1, l_1) \vee (t_1 = t_w \wedge l_1 = l)$
$ignored$ $Notify(v)$	$rv[v](t_s, l)$ $\wedge \neg\exists t_w.waiting(t_w, l)$	
$notifyAll(v)$	$rv[v](t_s, l)$ $\wedge\, \exists t_w.waiting(t_w, l)$	$\varphi_{waiting(t_1,l_1)} = waiting(t_1, l_1) \wedge (l_1 \neq l)$ $\varphi_{blocked(t_1,l_1)} = blocked(t_1, l_1) \vee (waiting(t_1, l_1) \wedge (l_1 = l))$
$ignored$ $NotifyAll(v)$	$rv[v](t_s, l)$ $\wedge \neg\exists t_w.waiting(t_w, l)$	
$blockLock(v)$	$\exists t \neq t_s.rv[v](t_s, l)$ $\wedge\, held\_by(l, t)$	$\varphi_{blocked(t_1,l_1)} = blocked(t_1, l_1) \vee (t_1 = t_s \wedge l_1 = l)$

Table 2.2: Operational semantics for concurrency statements. Actions above the two horizontal lines are non-blocking, the $blockLock(v)$ action is blocking.

$ignoredNotify(v)$ correspond to the possible behaviors when calling `v.notify()`: $notify(v)$ is enabled when there exists a thread waiting on the lock referenced by v, and the free variable t_w in its precondition corresponds to non-deterministic selection of the thread to be notified; $ignoredNotify(v)$ is enabled when no such thread exists. Similarly, $notifyAll(v)$ and $ignoredNotifyAll(v)$ model the behavior of `v.notifyAll()`. Technically, the translation of a Java statement (and condition) to several alternative actions can be performed by a front-end.

Formally, the meaning of actions is defined as follows:

Definition 2.3.3 *We say that* $C^\natural = \langle U, \iota \rangle$ ***rewrites into a configuration*** $C^{\natural\prime} = \langle U, \iota' \rangle$ *(denoted by* $C^\natural \Rightarrow_{ac} C^{\natural\prime}$*) where* ac *is an action, if there exists an assignment* Z *that satisfies the precondition of* ac *on* $C^\natural$*, and for every* $p \in P$ *of arity* k *and* $u_1, \ldots, u_k \in U$*,*

$$\iota'(p)(u_1, \ldots, u_k) =$$
$$[\![\varphi_p(v_1, v_2, \ldots, v_k)]\!]_2^{C^\natural}(Z[v_1 \mapsto u_1, v_2 \mapsto u_2, \ldots, v_k \mapsto u_k])$$

where $\varphi_p(v_1, \cdots, v_k)$ *is the formula for* p *given in Table 2.2.*

We say that a configuration $C^\natural$ ***transitively rewrites into a configuration*** $C^{\natural\prime}$ *(denoted by* $C^\natural \Rightarrow^* C^{\natural\prime}$*) if there exists a (potentially empty) sequence of configurations* $C^\natural = C_0^\natural, C_1^\natural, \ldots, C_n^\natural = C^{\natural\prime}$ *such that for each* $0 \leq i < n$ *,* $C_i^\natural \Rightarrow C_{i+1}^\natural$*.*

2.3.4 Safety Properties of Java Programs

Given a set of initial configurations $\mathcal{C}_0$, the set of *reachable* configurations $\mathcal{C}_R$ is the set of configurations that can be created by transitively rewriting a configuration from $\mathcal{C}_0$. More formally, a configuration $C_r \in \mathcal{C}_R$ iff there exists $C \in \mathcal{C}_0 . C \Rightarrow^* C_r$.

A safety property is formalized using logical formulae. We say that a safety property of a program *holds* if all reachable configurations satisfy the formula specifying the property.

Our analysis described in Section 2.4.1 aims at automatically verifying safety properties by guaranteeing to detect configurations where the properties are violated, if such configurations exist. Moreover, we sometimes also show that a liveness property at some reachable configuration holds by showing that a stronger safety property holds.

Table 2.3 lists some of the formulae used to detect configurations that violate a safety property. Formulae for other safety properties may be defined similarly.

In the Read-Write (RW) Interference formula, the first line states that both individuals t_r and t_w are different thread individuals, the second line states that thread t_r is at label lr and the thread t_w is at label lw, and the third line states that the variable x_w of thread t_w and variable x_r of thread t_r reference the same object o. The label lw is assumed to be a label of a statement with a writing access, and lr a label of a statement with a reading access.

Example 2.3.4 *In Fig. 2.3, the RW-Interference formula evaluates to* 0 *for the labels* lt_3 *(*`newHead = head.next`*) and* lp_6 *(*`tail.next = x_i`*) of the example program shown in Fig. 2.1(b). This is due to the fact that synchronization prevents the consumer thread* $\langle cns \rangle$ *from being at label* lt_3 *when the producer thread* $\langle prd \rangle$ *is at label* lp_6.

Even if synchronization was dropped, and the consumer and producer threads were allowed to be at lt_3 *and* lp_6 *correspondingly, RW-Interference would still evaluate to 0 since* `head` *and* `tail` *refer to different objects.*

The Write-Write (WW) Interference formula is similar to the RW Interference formula.

The Total Deadlock formula requires that for each thread t, there exists a lock l such that t is blocked on l. This is a strict formulation of the problem that can be generalized (e.g., allowing some thread to be in terminated state).

The Resource Ordering Criterion formula states that there exists a thread t holding a lock l_2, and blocked on a lock l_1 such that the ID of l_2 is greater than the ID of l_1.

The Nested Monitors formula states that o_{out} is a separation node in the configuration graph with respect to paths over the field `in`. Thus, every `in`-path from a node in the configuration graph reaching o_{in} passes through the node o_{out}. Therefore, a nested-monitors deadlock may be created when a thread becomes waiting on o_{in} while holding the lock of the object o_{out}.

The Missing Ownership formula states that there exists a thread t at label l_s which invokes `v.wait()` or `v.notify()` and does not hold the lock of the object l referenced by variable v.

2.4 An Abstract Program Model

The state-space exploration algorithm of Fig. 2.4 may be infeasible in programs with an unbounded number of objects. In this section we describe how to create a conservative representation of the concrete model presented in Section 2.3 in a way that provides both feasibility and high precision.

In Section 2.4.1 we use 3-valued logical structures to conservatively represent multiple configurations of a multithreaded program. Section 2.4.1 presents the concept of embedding, which is crucial for proving the correctness of our algorithm. Section 2.4.2 presents the abstract semantics derived from the concrete semantics presented in Section 2.3.3. Finally, Section 2.4.3 shows how to improve the precision of our analysis by adding instrumentation predicates.

2.4.1 Representing Abstract Program Configurations via 3-Valued Logical Structures

To make the analysis feasible, we conservatively represent multiple configurations using a single logical structure but with an extra truth-value $1/2$ denoting values which may be 1 and may be 0. The values 0 and 1 are called *definite values* whereas the value $1/2$ is called *indefinite value*. We allow an abstract

Formula	**Intended Meaning**
$\exists t_r, t_w : thread, o.(t_r \neq t_w)$ $\wedge\, at[lr](t_r) \wedge at[lw](t_w)$ $\wedge\, rv[x_w](t_w, o) \wedge rv[x_r](t_r, o)$	RW Interference between a thread (t_r) at label lr reading $x_r.fld$ and a thread (t_w) at label lw updating $x_w.fld$, where x_r and x_w are pointing to the same object o.
$\exists t_{w1}, t_{w2} : thread, o.(t_{w1} \neq t_{w2})$ $\wedge\, at[lw_1](t_1) \wedge at[lw_2](t_2)$ $\wedge\, rv[x_{w1}](t_{w1}, o) \wedge rv[x_{w2}](t_{w2}, o)$	WW Interference between a thread (t_{w1}) at label lw_1 writing $x_{w1}.fld$ and a thread (t_{w2}) at label lw_2 updating $x_{w2}.fld$, where x_{w1} and x_{w2} are pointing to the same object o.
$\forall t : thread.\exists l.blocked(t, l)$	Total Deadlock
$\exists t : thread, l_1, l_2.blocked(t, l_1)$ $\wedge\, held\_by(l_2, t) \wedge \neg idlt(l_2, l_1)$	Resource Ordering. A thread t is blocked on a lock "smaller" than a lock it is holding.
$\exists t_w : thread, o_{out}, o_{in}.waiting(t_w, o_{in})$ $\wedge\, held\_by(o_{out}, t_w) \wedge rv[in]^*(o_{out}, o_{in})$ $\wedge\, \forall o_p.((o_p \neq o_{out}) \wedge rv[in]^*(o_{out}, o_p)$ $\wedge\, rv[in]^*(o_p, o_{in})$ $\rightarrow \neg(\exists t_1, t_2.rv[in](t_1, o_p) \wedge rv[in](t_2, o_p))$	Nested Monitors. A thread t_w is waiting on an object o_{in} while holding the lock of an object o_{out} which structurally contains it, thus preventing any other thread from notifying t_w.
$\exists t.at[l_s](t) \wedge rv[v](t, l) \wedge \neg held\_by(l, t)$	Missing Ownership. Thread invoking `v.wait()` or `v.notify()` at label l_s when not holding the lock referenced by v.
See Section 2.5.2	Shared ADT
See Section 2.5.3	Thread Interactions

Table 2.3: Violations of safety properties detected in this chapter.

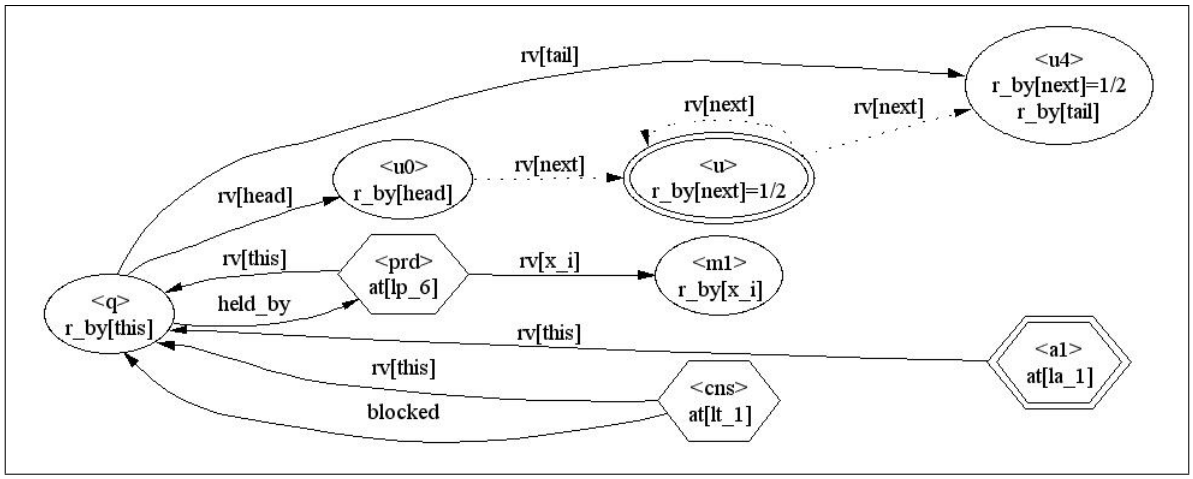


Figure 2.5: An abstract configuration $C_{2.5}$ *representing* the configuration $C^{\natural}_{2.3}$ shown in Fig. 2.3.

configuration to include *summary nodes*, i.e., individuals that represent one or more individuals in a represented concrete configuration. Technically, a summary node u has $\iota(eq(u,u)) = 1/2$.

Formally, an *abstract configuration* is a 3-valued logical structure $C = \langle U, \iota \rangle$ where:

- U is the universe of the 3-valued structure. Each individual in U represents possibly many allocated heap objects.
- ι is the interpretation function mapping predicates to their truth-value in the structure, i.e., for every predicate $p \in P$ of arity k, $\iota(p)\colon U^k \to \{0, 1/2, 1\}$. For example, $\iota(p)(u) = 1/2$ indicates that the truth value of p may be 1 for some of the objects represented by u and may also be 0 for some of the objects represented by u.

Embedding

We now formally define how configurations are represented using abstract configurations. The idea is that each individual from the (concrete) configuration is mapped into an individual in the abstract configuration. More generally, it is possible to map individuals from an abstract configuration into an individual in another less precise abstract configuration. The latter fact is important for our abstract transformer.

Formally, let $C = \langle U, \iota \rangle$ and $C' = \langle U', \iota' \rangle$ be abstract configurations. A function $f\colon U \to U'$ such that f is surjective is said to *embed* C *into* C' if for each predicate p of arity k, and for each $u_1, \ldots, u_k \in U$ one of the following holds:

$$\iota(p(u_1, u_2, \ldots, u_k)) = \iota'(p(f(u_1), f(u_2), \ldots, f(u_k)))$$

$$\textit{or}$$

$$\iota'(p(f(u_1), f(u_2), \ldots, f(u_k))) = 1/2$$

We say that C' *represents* C when there exists such an embedding f.

One way of creating an embedding function f is by using *canonical abstraction*. Canonical abstraction maps concrete individuals to an abstract individual based on the values of the individuals' unary predicates. All individuals having the same values for unary predicate symbols are mapped by f to the same abstract individual.

Example 2.4.1 *The abstract configuration* $C_{2.5}$ represents *concrete configuration* $C^{\natural}_{2.3}$.

We use dashed-edges to draw $1/2$*-valued binary predicates, and nodes with double-line boundaries to represent summary nodes.*

The summary node labeled a_1 *represents the threads* a_1, a_2, a_3 *which all have the same values for the unary predicates. The summary node labeled by* u *represents all queue items that are not directly referenced by the queue's head or tail. Note that the abstract configuration* $C_{2.5}$ *represents many configurations. For example, it represents any configuration with 3 or more queue items. In a similar fashion, the abstract configuration represents configurations with one or more threads that reside at label* la_1.

Note that the RW-Interference condition evaluates to 0 *over the abstract configuration* $C_{2.5}$.

2.4.2 An Abstract Semantics

We use the same simple algorithm from Fig. 2.4 for exploration of the abstract state space. The operations used by the algorithm are modified to work for abstract configurations. The *rewrites* relation is modified to conservatively model the effect of an action on the given abstract configuration (possibly representing multiple configurations). In addition, the state-space exploration now starts with $\mathcal{C}_0$ being the abstraction of initial configurations.

Implementing an algorithm for computing the *rewrite* relation on abstract configurations is non-trivial because one has to consider all possible relations on the set of represented (concrete) configurations.

The *best conservative effect* of an action (also known as the *induced effect* of an action) [25] is defined by the following 3-stage semantics: (i) A concretization of the abstract configuration is performed, resulting in all possible configurations *represented* by the abstract configuration; (ii) The action is applied to each resulting configuration; (iii) Abstraction of the resulting configurations is performed, resulting in a set of abstract configurations *representing* the results of the action.

Our prototype implementation described in Section 2.6 operates directly on abstract configurations, and obtains actions which are more conservative than the ones obtained by the best transformers. Our experience shows that these actions are still precise enough to detect violations of the safety properties as listed in Table 2.3, without producing *false alarms* on our example programs.

Definition 2.4.2 *We say that an abstract configuration* C ***rewrites into an abstract configuration*** C' *(denoted by* $C \Rightarrow_{ac} C'$*) where* ac *is an action, if for* C *and for* C' *there exists* $C^{\natural}$ *and* $C^{\natural\prime} = \langle U^{\natural}, \iota^{\natural\prime} \rangle$

such that: (i) $C^\natural$ is in the concretization of C, i.e., C represents *$C^\natural$, (ii) C' is the* canonical abstraction *of $C^{\natural\prime}$, (iii) there exists an assignment Z that satisfies the precondition of ac on $C^\natural$, and for every $p \in P$ of arity k and $u_1, \ldots, u_k \in U^\natural$,*

$$\iota^{\natural\prime}(p)(u_1, \ldots, u_k) =$$
$$[\![\varphi_p(v_1, v_2, \ldots, v_k)]\!]_3^C(Z[v_1 \mapsto u_1, v_2 \mapsto u_2, \ldots, v_k \mapsto u_k])$$

where $\varphi_p(v_1, \cdots, v_k)$ is the formula for p given in Table 2.2. We write $C \Rightarrow C'$ if for some action ac $C \Rightarrow_{ac} C'$.

Example 2.4.3 *The abstract configuration $C_{2.6,0}$ shown in Fig. 2.6 represents an unbounded number of threads all at label la_1. The actions for label la_1 are $lock(this)$ and $blockLock(this)$.*

The infinite set of configurations $\{C_{2.6,0,1}, C_{2.6,0,2}, \ldots\}$ is the set of (concrete) configurations after concretization. After concretization the preconditions of the actions are evaluated. The precondition for $lock(v)$ evaluates to 1 and the precondition for $blockLock(v)$ evaluates to 0. Thus $lock(v)$ is applied. The infinite set of configurations $\{C_{2.6,1,1}, C_{2.6,1,2}, \ldots\}$ is the set after the application of $lock(v)$. The set of abstract configurations $\{C_{2.6,2,1}, C_{2.6,2,2}\}$ is the finite set of configurations after abstraction.

The membership operator $member(C, stateSpace)$ of Fig. 2.4 can be modified to check if the configuration C is already represented by one of the configurations in $stateSpace$. This is an optimization for preventing exploration of redundant configurations.

2.4.3 Instrumentation

Instrumentation predicates record derived properties of individuals. Instrumentation predicates are defined using a logical formula over core predicates. Updating an instrumentation predicate is part of the predicate-update formulae of an action.

The information recorded by an instrumentation predicate in a configuration may be more precise than evaluating the defining formula of the instrumentation predicate over the configuration. This is known as the *Instrumentation Principle* introduced in [91].

The mapping of individuals in a configuration into an abstract individual of an abstract configuration is directed by the values of the unary predicates. By adding unary instrumentation predicates, one may allow finer distinction between individuals, and thus may improve the precision of the analysis.

Example 2.4.4 *Consider an unbounded number of threads competing to acquire a single shared lock. Assume that a thread t_1 has already acquired the lock. The configuration $C_{2.7,0,1}$ shown in Fig. 2.7 corresponds to a state in which some thread tried to acquire the lock and consequently became blocked on the lock. In this configuration, the formula $\exists t, l.rv[this](t, l) \wedge blocked(t, l)$ evaluates to $1/2$. Configuration $C_{2.7,0,2}$ shows the same global state when the instrumentation predicate $is\_blocked(t)$ is used.*

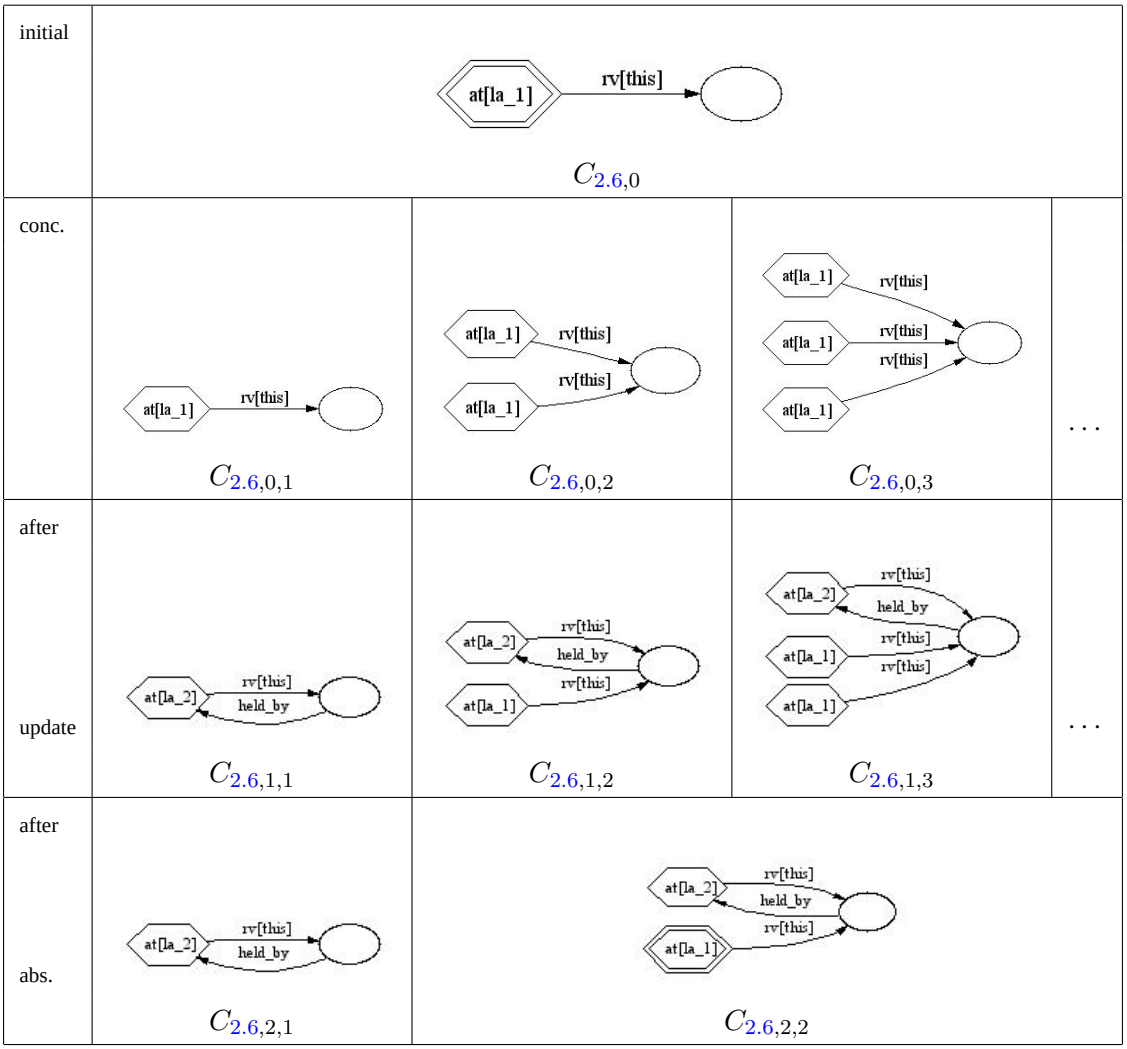


Figure 2.6: Concretization and predicate-update for an unbounded number of threads all performing the $approveHead()$ method of the running example.

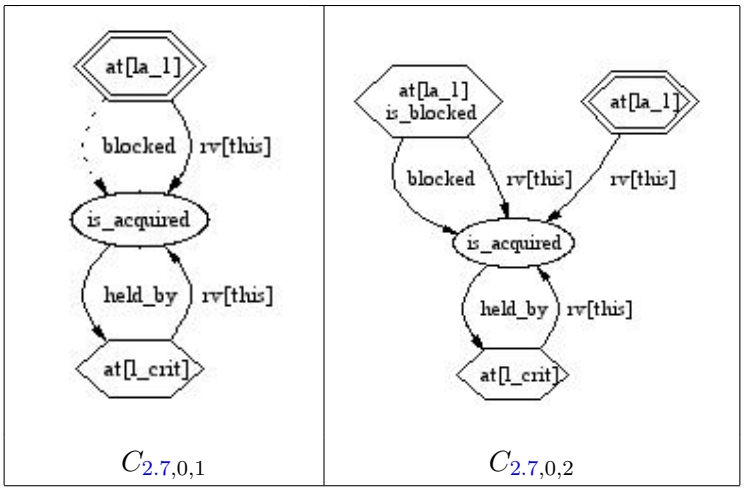


Figure 2.7: Instrumentation predicate $is\_blocked(t)$.

Now, one can check the existence of a blocked thread using the stored value of the instrumentation predicate $is\_blocked(t)$*, which evaluates to* 1*. Note that in this case evaluation of the original formula over the configuration with instrumentation also evaluates to* 1 *rather than to* $1/2$*, but this is not always the case.*

2.5 Verifying Safety Properties

We use the instrumentation predicates listed in Table 2.4 to improve the precision of our analyses. The following sections list a more precise formulation of the formulae of Table 2.3 by using instrumentation predicates whenever possible.

2.5.1 Deadlock

We use the $wait\_for(t_1, t_2)$ instrumentation predicate to detect a cyclic $wait\_for$ dependency. We use $slock(t)$ to track the resource-ordering local property for each thread. Thus, the resource ordering violation can be formulated as $\exists t.slock(t)$. The definition of $slock(t)$ uses the predicate $lt[id](v_1, v_2)$ which records the order between locks according to the value of their `id` fields. Each lock object is assumed to have a unique id recorded in its `id` field (e.g., such an id could be provided using the `java.lang.Object`'s $hashCode()$ method). The predicate $lt[id](l_1, l_2)$ is true when the id of l_1 is less than the id of l_2. The order between objects can be used for deadlock prevention by breaking cyclic allocation requests [97].

The formula for nested-monitors deadlock is given below:

$$\exists t_w : thread, o_{out}, o_{in}.waiting(t_w, o_{in}) \wedge held\_by(o_{out}, t_w) \wedge rf[in](o_{out}, o_{in})$$
$$\wedge \forall o_p.((o_p \neq o_{out}) \wedge rf[in](o_p, o_{in}) \wedge rf[in](o_{out}, o_p) \rightarrow \neg is[in](o_p))$$

Predicate	**Intended Meaning**	**Defining Formula**
$is[fld](l_1)$	l_1 is referenced by the field fld of more than one object	$\exists t_1, t_2.(t_1 \neq t_2) \rightarrow rv[fld](t_1, l_1) \wedge rv[fld](t_2, l_2)$
$r\_by[fld](l)$	l is referenced by the field fld of some object	$\exists o.rv[fld](o, l)$
$lt[ifld](v_1, v_2)$	the value of `ifld` of v_1 is less than that of v_2	$\exists i_1, i_2.ival[ifld](v_1, i_1) \wedge ival[ifld](v_2, i_2) \wedge succ^*(i_1, i_2)$
$is\_acquired(l)$	l is acquired by a thread	$\exists t.held\_by(l, t)$
$is\_blocked(t)$	t is blocked on a lock	$\exists l.blocked(t, l)$
$is\_waiting(t)$	t is waiting on a lock	$\exists l.waiting(t, l)$
$slock(t)$	t violates the resource ordering criterion	$\exists l_1, l_2.is\_thread(t) \wedge blocked(t, l_1) \wedge held\_by(l_2, t) \wedge \neg lt[id](l_2, l_1)$
$wait\_for(t_1, t_2)$	t_1 is waiting for a resource held by t_2	$\exists l_b.blocked(t_1, l_b) \wedge held\_by(t_2, l_b)$
$rf[fld](o_1, o_2)$	object o_2 is reachable from object o_1 using a path of fld edges	$rv[fld]^*(o_1, o_2)$
$rt[ref, fld](t, o)$	object o is reachable from thread t by a path starting with a single ref edge followed by any number of fld edges	$\exists o_t.rv[v](t, o_t) \wedge rv[next]^*(o_t, o)$

Table 2.4: Instrumentation predicates for partial Java semantics.

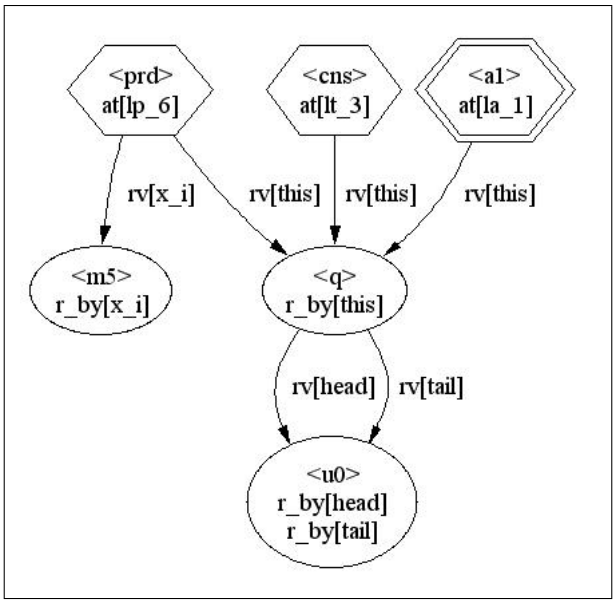


Figure 2.8: An abstract configuration $C_{2.8}$ in which interference between the consumer and the producer is detected.

2.5.2 Shared Abstract Data Types

We define a set of reachability predicates similar to the ones defined in [91]. We use the reachability information to define invariants for ADT operations. For example:

- At the end of a put operation — the new item is reachable from the head of the queue.
- At the end of a $take$ operation — the taken item is reachable from the taking thread and is no longer reachable from the head of the queue.

2.5.3 Thread State Errors

We use instrumentation predicates to record thread-state information: $ts\_created(t)$, $ts\_running(t)$, $ts\_blocked(t)$, $ts\_waiting(t)$ and $ts\_dead(t)$. In order to identify thread-state errors, we add preconditions identifying when an action is illegal or suspicious. These preconditions are listed in Table 2.5.

Example 2.5.1 *Assume an erroneous version of the running example (Fig. 2.1) in which an unsynchronized version of* `put()` *is used. Configuration* $C_{2.8}$ *shown in Fig. 2.8 demonstrates a possible interference in the program identified by our analysis. In the configuration* $C_{2.8}$ *a consumer is trying to* `take()` *the last item, and a producer is simultaneously trying to* `put()` *an item.*

Problem	**Action**	**Precondition**	**Warning**
Multiple starts	$v.start()$	$rv[v](t_r, dt) \wedge ts\_running(dt)$	$IllegalThreadStateException$
		$rv[v](t_r, dt) \wedge ts\_dead(dt)$	Dead thread cannot be re-started
Premature stop	$v.stop()$	$rv[v](t_r, dt) \wedge ts\_created(dt)$	Thread stopped before started
Missing ownership	$v.wait()$	$rv[v](t_r, l) \wedge \neg held\_by(l, t)$	$IllegalMonitorStateException$
	$v.notify()$	$rv[v](t_r, l) \wedge \neg held\_by(l, t)$	$IllegalMonitorStateException$
		$rv[v](t_r, l) \wedge \neg\exists t_w.waiting(t_w, l)$	A notify was ignored
Premature join	$v.join()$	$rv[v](t_r, dt) \wedge ts\_created(dt)$	Thread join before started
Late setDaemon	$v.setDaemon()$	$rv[v](t_r, dt) \wedge ts\_running(dt)$	$IllegalMonitorStateException$

Table 2.5: Preconditions for checking illegal and suspicious thread interactions.

The consumer thread reached label lt_3 and is about to execute the action for `newHead = head.next`. *The producer thread, having found that the queue is not empty, reached label lp_6, and is about to execute the* `tail.next=x_i` *action. The RW-Interference formula from Table 2.3 evaluates to* 1 *for this configuration since both threads reference the same object $< u0 >$. Thus RW-Interference is detected.*

It is important to note that if the queue has more than one item, RW-Interference is not introduced, and our analysis will report that RW-Interference does not occur (since `head` *and* `tail` *refer to different objects).*

2.5.4 Unbounded Number of Threads

When a system consists of many identical threads, the state-space can be reduced by exploiting symmetry.

In model checking, the global state of a system is usually described as a tuple containing thread program-counters, and value assignments for shared variables [38]. In [38], symmetry is found between process indices. In our framework, thread names are only determined by thread properties. Thus, there is no need to explicitly define permutation-equivalence for symmetry reduction. *The mapping to the canonic names eliminates symmetry in the abstract state space.*

We demonstrate the power of our abstraction by taking the example of a critical section from [38], and verifying that the *mutual exclusion* property holds for an *unbounded number of threads*.

Example 2.5.2 *Consider the $approveHead()$ method of class Queue. We would like to verify mutual exclusion over the critical section protected by* `synchronized(this)`. *For readability of this ex-*

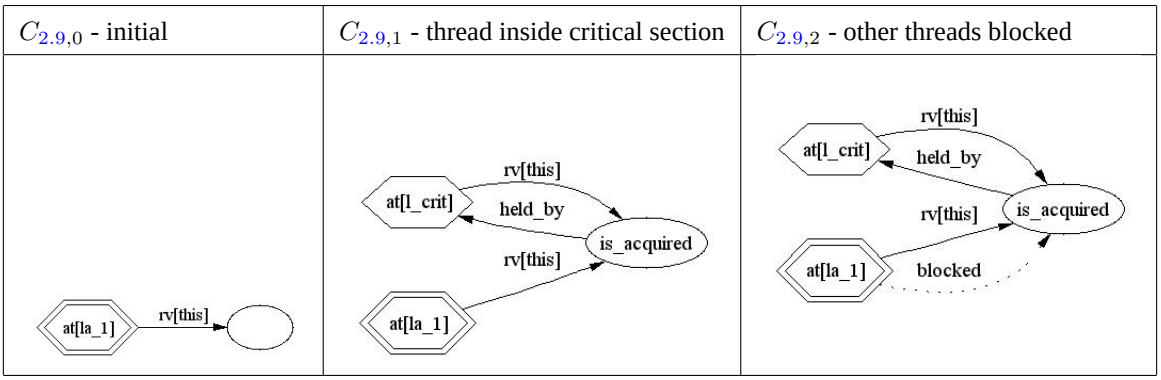


Figure 2.9: Configurations arising in mutual exclusion with an unbounded number of threads.

ample we define all labels inside the critical section as a single label l_{crit}. The property we detect is $\exists t_1, t_2.(t_1 \neq t_2) \wedge at[l_{crit}](t_1) \wedge at[l_{crit}](t_2)$). The initial state for the analysis contains an unbounded number of threads *represented by a summary node. Fig. 2.9 shows three important abstract configurations arising in the analysis of the example.*

In addition, using thread names that are only determined by thread properties reduces the number of equivalent interleavings that have to be considered. For example, consider a program with five threads, each performing a single assignment to a local boolean variable b initialized to false, setting its value to true. That is, each thread executes the single statement l_1 `b = true;` l_2. When the program terminates, the local boolean variable b of each thread is set to true. Analyzing this program with explicitly named threads will result with 125 possible interleavings that have to be considered (see Fig. 2.10). Analyzing the program in our approach will only consider a single (representative) interleaving (see Fig. 2.11).

2.6 Prototype Implementation

In this section, we briefly describe our prototype implementation and present experimental results of applying the framework on a few small but interesting example programs. More elaborate experimental results for the verification of concurrent queue algorithms are provided in Chapter 7.

We have implemented a prototype of our framework called 3VMC [113]. Our implementation is based on the 3-valued logic engine of TVLA [64]. We applied the analyses to several small but interesting programs. Table 2.6 summarizes the programs we tested, with the number of configurations created, and running times. Running times were measured using Sun's JVM1.2.2 for Windows NT, running on a 600MHZ Pentium III.

It is important to note that the cost of verification for an unbounded number of threads in our approach is doubly exponential in the number of predicates, while the cost of verification with explicit

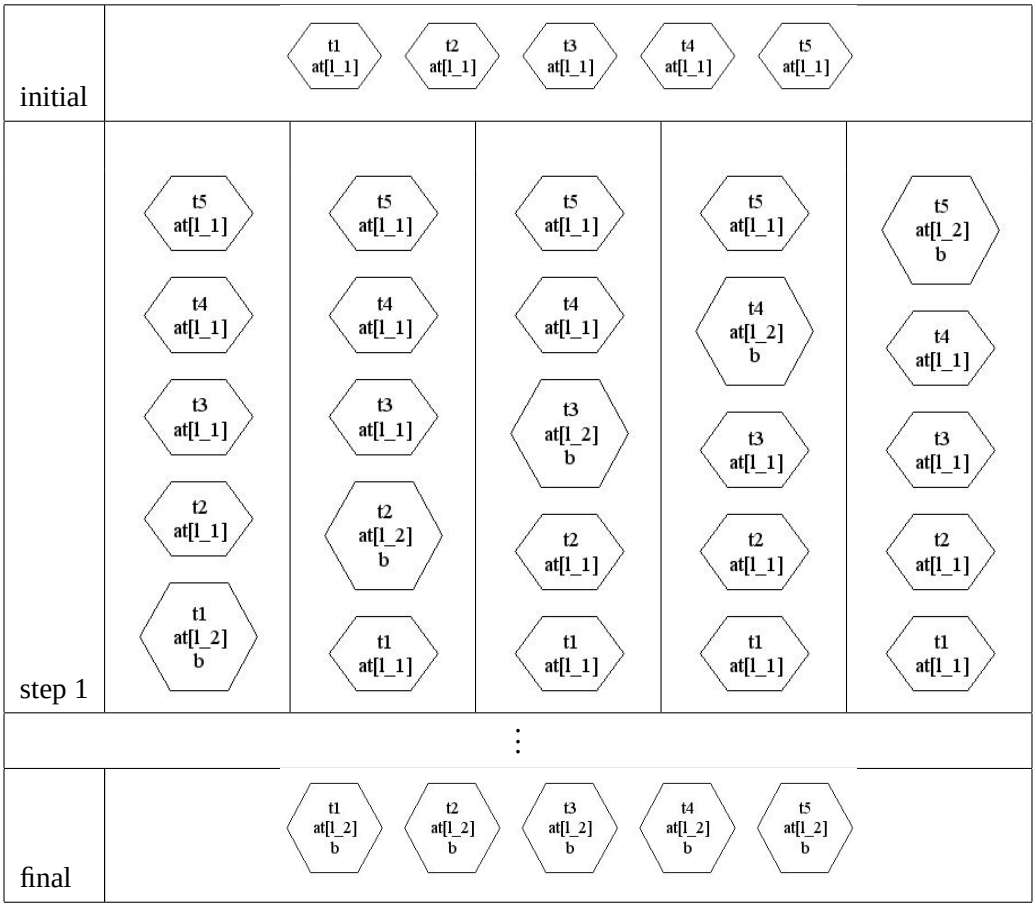


Figure 2.10: Configurations arising with explicit thread names.

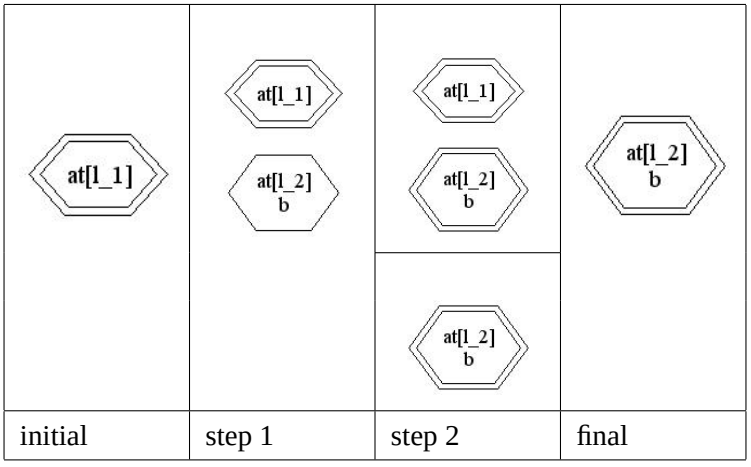


Figure 2.11: Configurations arising with canonical thread names.

Program	Description	Properties	Config.	Time
swap	swap list elements	absence of data races and deadlock	16	10
swap_ord	swap list elements with resource ordering	absence of data races and deadlock	1	12
stack	non-synchronized stack	absence of data races	184	304
sStack	synchronized stack	absence of data races	104	330
mutex	mutual exclusion with unbound threads	mutex	33	2
nestedMon	nested monitors	absence of deadlock	42	7
prodCons	producer consumer	absence of data races	416	68
sProdCons	synchronized producer consumer	absence of data races	195	48
DP	dining philosophers unbound threads	absence of deadlock	514	23

Table 2.6: Number of configurations, and running times in seconds for the programs analyzed.

thread names is exponential in the number of threads. As a result, verifying a property for an unbounded number of threads is not only stronger, but sometimes more efficient than verifying the property for an a priori bounded number of threads. For example, verifying mutual exclusion for the mutex program with 5 explicitly named threads takes over 70 seconds, whereas verification for an unbounded number of threads takes only 2 seconds.

In our prototype, the conservative effect of an action is implemented in terms of the $focus$ and $coerce$ operations (see [91] for more details). The soundness of our implementation is guaranteed by a generalization of the embedding theorem of [91] for infinite concrete configurations (see proof in Appendix B.1.1).

The $swap$ and $swap\_ord$ programs use two threads swapping items in a linked list. $swap$ does not use resource ordering, and thus may deadlock, $swap\_ord$ uses resource ordering, and thus cannot deadlock. $stack$ and $sStack$ are non-synchronized and synchronized versions of a Stack ADT manipulated by multiple threads. $mutex$ is a simple program using mutual exclusion to protect a critical section. $prodcons$ and $sProdCons$ are implementations of Queue ADT manipulated by producer and consumer threads. The DP program is an implementation of the *dining philosophers* problem with an unbounded number of philosopher threads.

While these example programs are small, the scenarios they explore are rather complicated (e.g., nested monitors). We are encouraged by the fact that for these examples, our analysis concluded with no false alarms. In Chapter 7, we explore more realistic example programs.

Chapter 3

Property-Guided Abstraction

In this chapter, we consider the problem of *typestate verification* for *shallow* programs; i.e., programs where pointers from program variables to heap-allocated objects are allowed, but where heap-allocated objects may not themselves contain pointers. We prove a number of results relating the complexity of verification to the nature of the finite state machine used to specify the property. Some properties are shown to be intractable, but others which appear to be quite similar admit polynomial-time verification algorithms. Our results serve to provide insight into the inherent complexity of important classes of verification problems. In addition, the program abstractions used for the polynomial-time verification algorithms may be of independent interest.

In solving a problem of this sort, the grand thing is to be able
to reason backward. ... In the everyday affairs of life
it is more useful to reason forward.
–Sir Arthur Conan Doyle, *A Study in Scarlet*.

3.1 Introduction

The desire for more reliable software has led to increasing interest in extended static checking: statically verifying whether a program satisfies certain desirable properties. A technique that has received particular attention is that of finite state or *typestate* verification (e.g., see [103, 102, 77, 21, 29, 6, 30, 44, 43, 60, 4]). In this model, objects of a given type exist in one of finitely many *states*; the operations permitted on an object depend on the state of the object, and the operations may potentially alter the state of the object. The goal of typestate verification is to statically determine if the execution of a given program may cause an operation to be performed on an object in a state where the operation is not permitted.

Typestate verification can be used to check that objects satisfy certain kinds of temporal properties; e.g., that an object is not used before it is initialized, or that a file is not used after it is closed. In this

chapter, we will specify such properties using regular expressions or finite state automata that define the set of *valid* sequences of operations that can be performed on an object.

Our goal in this chapter is to develop an initial understanding of how the difficulty of performing typestate verification relates to the *nature of the property being verified*. Among other things, we will show that not all finite state properties are equally hard to verify. For example, given a *shallow* program (where pointers from program variables to heap-allocated objects are allowed, but where heap-allocated objects may not themselves contain pointers), we show that verifying that a file is not read after it is closed can be done in *polynomial time*, while verifying that a file is not read before it is opened is *PSPACE-Complete*.

While there has been much progress in many aspects of automated program verification, we are not aware of any previous work relating the difficulty of typestate verification to properties of the finite state automaton. This work is part of a broader effort to develop efficient program verification techniques that are tailored to the property being verified [84].

Typestate Verification and Shallow Programs

In order to meaningfully compare the complexity of verification algorithms, we need to make some baseline assumptions about the precision of the analysis. In this chapter, we will use the term *verification* to mean verification that is *precise* modulo the widely-used assumption that all paths in the program are feasible. Specifically, given a finite state property, a path in a program is said to be an *error path*, if execution along that path would cause an invalid sequence of operations to be performed on at least one *object* and the goal of typestate verification is to determine if a given program has any error path.

Typestate verification can be done in polynomial time if the program to be verified allows no inter-variable aliasing. Conversely, it is a straightforward consequence of previous results [61, 74] that if a program has *two or more* levels of pointers, typestate verification is PSPACE-hard[1]. In this chapter, we therefore concentrate on understanding the class of *shallow* programs occupying a point in between these extremes.

Assume we wish to perform typestate verification for objects of a type T. A *T-shallow* program is a well-typed procedure-free program where all variables are pointers to T-typed objects, and whose statements are allocations (creation of a new object of type T), copy assignments (copying the value of a variable to another), or invocations of an operation on a variable. Note that shallow programs may contain multiple pointers to objects of type T, but allocated objects may not themselves contain T-pointers. In other words, pointers in shallow programs are *single-level* [74]. Our results also apply to programs that manipulate complex or recursive types where allocated objects contain pointers, *provided that those pointers cannot refer to objects of type T*. Programs that are shallow with respect to a given

[1] In the presence of recursive data structures, typestate verification is undecidable [62, 83].

type, e.g. `File`, are not uncommon in practice.

Example: Verifying File Operations

Consider the problem of checking that a closed file is never read or closed again, which we will refer to as `read`*;`close`. In general, we will use regular expressions to designate sequences of *valid* operations on an object of a given type, where a sequence is valid iff it is a prefix of a string in the language defined by the regular expression.

The principal difficulty in doing precise verification arises from determining how *aliasing* interacts with operations on objects. Some prior work on typestate verification (e.g. [27]) has employed a two-step approach to the problem, in which an initial phase performs a conservative heap analysis of the program, and a subsequent phase uses the information from the heap analysis to do typestate analysis. However, we can see from the program fragments in Figure 3.1 that such an approach can sometimes lead to imprecise results. One can easily verify that in both Figures 3.1(a) and 3.1(b), all sequences of file operations on a given object are prefixes of `read`*;`close`; i.e., that no `read` ever follows a `close`.

However, consider a two-phased approach in which the heap analysis is separate from the typestate analysis. In Fig. 3.1(a), a precise (and correct) heap analysis will determine that program variable `z` at program point `s2` may point to the object created at `s0` or the object created at `s1`. Furthermore, a precise typestate analysis will determine that the object created at `s1` could be in a *closed* state at `s2`. A two-phased analysis must therefore erroneously conclude that the read could be performed on a closed file. Similarly, in Fig. 3.1(b), any conservative heap analysis would determine that objects created at program points `s3` and `s5` could reach the read statement at `s4`. In addition, a typestate analysis would also determine that the objects created at program points `s3` and `s5` could be in a closed state at `s4`. The analysis would, however, not be able to discover that `f` can never point to a closed object at `s4`, and would incorrectly indicate a possible error. In this chapter we show that for a certain class of problems (including `read`*;`close`), it is possible to formulate a precise polynomial-time verification algorithm for shallow programs.

Main Results

The main complexity results established in this chapter are as follows (in all cases except the last one, we assume that programs are shallow):

- Verification is in P for omission-closed properties: a property is said to be omission-closed if every subsequence of a valid sequence is also a valid sequence. (Example: `read`*;`close`.)
- Verification is NP-Complete for acyclic programs (i.e., programs without loops) and PSPACE-complete for arbitrary programs for properties with a repeatable enabling sequence: a property

```
s0 : x := new ();
s1 : y := new ();
z := y;
if (?) {
  y.close();
  z := x;
}
s2 : z.read();
```

(a)

```
s3 : f := new ();
while (?) {
  s4 : f.read();
  if (?) {
    f.close();
    s5 : f := new ();
  }
}
```

(b)

Figure 3.1: Program fragments illustrating the effect of aliasing on typestate verification.

is said to have a repeatable enabling sequence if there is an automaton state where a particular sequence γ of operations is invalid, but sequences of the form $\beta^+\gamma$ are valid for some β. Example: open$^+$; read.

- An integer-valued function f is said to be a bound on the shortest error path length for a typestate property if every erroneous program of size n is guaranteed to have an error path of length $f(n)$ or less. If PSPACE is not equal to NP, then no polynomial bound exists for the shortest error path length for properties with a repeatable enabling sequence. (In other words, it may not be possible to find short, i.e., polynomial size error paths in the worst case.)

- Verification is in P for acyclic programs for almost-omission-closed properties: a property is said to be almost-omission-closed if there is an integer k such that every subsequence of a valid sequence of length greater than k is also valid. Example: open; read. Note that any property with only finitely many valid sequences is trivially almost-omission-closed.

- Verification is in P for almost-omission-closed properties that have a polynomial bound on the shortest error path length.

- A program is said to have a maximum aliasing width of k if there is no path in the program that will produce an object pointed to by more than k different variables. Arbitrary finite state properties for programs of size n with a maximum aliasing width of k may be verified in time $O(n^{k+1})$ for programs of size n.

- Alias analysis and typestate verification are NP-hard for programs with maximum aliasing width of three and aliasing depth of two. (A program is said to have aliasing depth of two if the program contains pointers to pointers).

	Example	Definition	Acyclic Programs (Shallow)	Cyclic Programs (Shallow)	Bounded Aliasing Width	
					Shallow	Non-shallow
Ommission-Closed	`read`*;`close`	$\forall\alpha\beta\gamma.$ *Valid*$(\alpha\beta\gamma)$ $\Rightarrow$ *Valid*$(\alpha\gamma)$	P	P	P	NP-hard
Almost-Ommission-Closed	`open`;`read`	$\exists k\forall\alpha\beta\gamma.$ $(\lvert\alpha\beta\gamma\rvert \geq k \wedge$ *Valid*$(\alpha\beta\gamma))$ $\Rightarrow$ *Valid*$(\alpha\gamma)$	P	Poly. Err. Path General: ?		
Repeatable Enabling Sequence	`open`$^+$;`read`	$\exists\alpha\beta\gamma.$ *Valid*$(\alpha\beta^+\gamma)\wedge$ $\neg$*Valid*$(\alpha\gamma)$	NP complete	PSPACE complete		
Other	(`lock`;`unlock`)*		?	?		

Figure 3.2: An overview of our complexity results.

The results above are summarized in Fig. 3.2 in terms of the properties of regular expressions which define the properties to be verified (the notation used there will be defined in Section 3.2).

The polynomial-time verification results summarized above use program abstractions that may be of independent interest—in particular, they may prove useful as the starting point for developing more general abstractions for non-shallow programs (e.g., in a manner similar to [84]). The bulk of the abstractions we use are *predicate abstractions* [49]; however we show in the sequel that the choice of predicates used in a predicate abstraction can have a dramatic impact on the efficiency of the resulting analysis. Our predicate vocabularies are carefully designed to yield efficient analyses without sacrificing precision. In addition, in Section 3.5, we develop a novel *integer* abstraction, which is based on *counting* the number of program paths along which a simple property holds true; this in turn allows inferring whether a more complex property holds.

Related Work

There has been significant recent interest in a variety of property verification techniques, many of them focusing on typestate verification. While significant progress has been made in improving the precision and efficiency of verification, developing verification techniques that are sufficiently precise and scalable to handle industrial-size applications for a wide variety of problems is still a challenge, and motivates

our work here.

One of the open challenges in typestate verification is an adequate treatment of aliasing. Some approaches avoid the issue: e.g., the original work on typestate verification [103, 102] did not allow any aliasing; more recent work on typestate verification based on linear types [29] also restricts aliasing severely. Other approaches (e.g. [27]) perform alias analysis and typestate verification separately: an initial phase performs a conservative alias analysis for the program, and a subsequent phase uses the information from the alias analysis to do typestate verification. However, this can lead to imprecise results, as illustrated by the examples in Fig. 3.1.

A second challenge to practical verification is dealing with infeasible program paths (e.g., [54, 10]). Das et al. [27] address this issue using efficient path-sensitive algorithms (which eliminate certain infeasible paths from consideration during analysis), but do not track certain additional information, e.g., aliasing, precisely. Our algorithms do not address the question of path sensitivity, but there could be merit in combining aspects of our approach with those that eliminate infeasible paths.

One of the primary intuitions behind the algorithms presented in this chapter (for shallow programs) is that maintaining just the right correlation required between "analysis facts" can be the key to efficient and precise verification: maintaining no correlations (independent attribute analysis) can lead to imprecision, while maintaining all correlations (relational analysis) can lead to inefficiency. Chapter 6 shows one way to exploit this intuition for verification of arbitrary (i.e. non-shallow) programs as well.

Several recent verification approaches [5, 53] combine predicate abstraction [49], counterexample-guided refinement of the predicate vocabulary [15], and exploration of the resulting abstract state space using model-checking. These techniques use symbolic and theorem-proving techniques to identify a set P of predicates relevant to the problem of interest, then model-check the resulting finite state system over a state space constructed from the powerset lattice $2^{P \to \{true, false\}}$. This process iterates with increasingly larger sets of predicates until a satisfactory result is obtained. In principle, these algorithms have the potential to avoid imprecision due to both aliasing and path infeasibility. However, the worst-case complexity of a *single* iteration is exponential in the number of predicates. By contrast, while most of the algorithms we present are based on abstractions by a set of predicates Q, our analysis is based on the function-space lattice $Q \to \{false, maybe\}$, and runs in time linear in the size of Q. This approach yields polynomial-time algorithms, while none of the techniques based on model-checking have a polynomial-time worst-case complexity for the same problems (even though they may utilize a smaller number of predicates than our algorithm). Our selection of predicates ensures that the use of the smaller function space lattice results in no loss of precision, i.e., we ensure that our abstraction is *complete* (e.g., see [46]). Finally, the predicate abstractions we use are dependent solely on the nature of the typestate problem being verified, and do not require expensive predicate discovery at verification time.

Finally, we note that our lower bound results follow the tradition set by earlier complexity results

due to Landi and Ryder [61] and Muth and Debray [74].

3.2 Terminology and Notation

In this section, we provide some basic definitions that we will use in the rest of the chapter.

Definition 3.2.1 (Shallow Program) *A **shallow program** is a* `<Stmt>` *defined by the following context-free grammar, where the* `?` *denotes a nondeterministic branch (i.e., an uninterpreted conditional). All variables* `<Var>` *in the language are references to objects of type T. All operations* `<Op>` *in the language are methods supported by type T.*

```
<Stmt> ::= <Var> := <Var> |  <Var> := new() |  <Var>.<Op>()
         |  <Stmt>;<Stmt> |  if (?) <Stmt> [ else <Stmt> ]
         |  Label: <Stmt> |  goto Label
```

We will make the simplifying assumption that when a program begins execution all program variables point to separate objects (i.e., initialized to non-aliased values), and all objects reside in their initial state. In other respects, the semantics of shallow programs is completely standard, and we will not formalize it here. We will, however, appeal to the intuitive notion of a *path* ρ through a program P (or P-path): a valid sequence of statements starting at P's entry.

In this chapter, we will study safety properties of shallow programs. Although safety properties could be specified via temporal logics (e.g., LTL [17]), we will use finite automata or regular expressions to simplify the presentation. Formally:

Definition 3.2.2 (Prefix-Closed Safety Automaton) *A **prefix-closed safety property** $\mathcal{F}$ is represented by a finite state automaton (FSA) $\mathcal{F} = \langle \Sigma, Q, \delta, \text{init}, Q \setminus \{\text{err}\} \rangle$ where Σ is the automaton alphabet consisting of observable operations, Q is the set of automaton states, δ is the transition function mapping a state and an operation to a successor state,* init $\in Q$ *is a distinguished* initial state, err $\in Q$ *is a distinguished* error state *for which for every $\sigma \in \Sigma$, $\delta(\text{err}, \sigma) =$* err*, and all states in $Q \setminus \{\text{err}\}$ are accepting states. We say that q' is the successor of a state q on operation* `op` *when $\delta(q, \texttt{op}) = q'$. Given a sequence of operations $\alpha = \texttt{op}_1; \texttt{op}_2; \ldots; \texttt{op}_k$, we write Valid$_{\mathcal{F}}(\alpha)$ or $\alpha \in$ Valid$_{\mathcal{F}}$ when α is accepted by $\mathcal{F}$, and we write Invalid$_{\mathcal{F}}(\alpha)$ when α is not accepted by $\mathcal{F}$.*

For brevity, we will refer to safety properties using a regular expression representing the language accepted by an automaton, rather than specifying the automaton itself. When specifying a safety property using a regular expression, we will adopt the convention that a regular expression α denotes the *prefix*

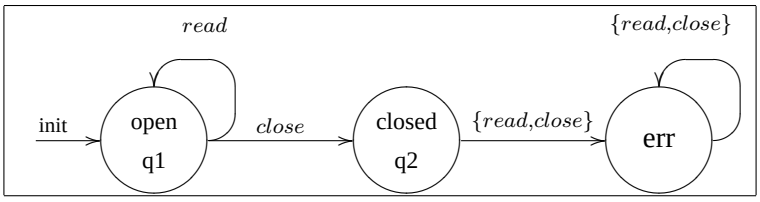


Figure 3.3: A finite-state automaton for the property `read`*;`close`.

closure of the set of sequences of operations defined by α. For example, when we write `read`*;`close` we also consider ϵ (the empty sequence) and `read` to be valid sequences.

Example 3.2.3 *Consider the property* `read`*;`close` *stating that a file may be read an arbitrary number of times before it is closed (and should never be read after it was closed and never be closed twice). The alphabet for this problem consists of two operations* $\Sigma = \{\texttt{read}, \texttt{close}\}$. *The FSA for this property is shown in Fig. 3.3.*

When verifying a safety property represented by an automaton $\langle Q, \mathit{init}, \mathit{err}, \Sigma, \delta \rangle$ for a shallow program P, we will assume that each method name used in P is mapped to an element of Σ. Given this convention, we will use names of operations in Σ and methods in P interchangeably, i.e., we will say that a statement of the form `x.op()` invokes an operation `op` $\in \Sigma$. We can then relate method invocations to sequences of operations in Σ as follows:

Definition 3.2.4 (Operation Sequences for Objects) *Given a P-path ρ, $\mathcal{U}(\rho)$ denotes the set of object instances created during this execution, and for any object $o \in \mathcal{U}(\rho)$, $\rho[o]$ denotes the sequence of operations performed on o during execution of ρ.*

Given the definitions above, we can now formally describe the class of verification problems we wish to solve:

Definition 3.2.5 (SV$_\mathcal{F}$) *Given a safety property $\mathcal{F}$, the **shallow verification problem** for $\mathcal{F}$, SV$_\mathcal{F}$, determines for any shallow program P whether there exists a path P-path ρ such that $\rho[o] \in \mathit{Invalid}_\mathcal{F}$ for some $o \in \mathcal{U}(\rho)$.*

3.3 Omission-Closed Properties in Polynomial Time

In this section, we show that *omission-closed* properties can be verified in polynomial time.

Omission-Closed Properties

Informally, a property is omission-closed if the set of all valid sequences of operations is closed with respect to omissions: any sequence obtained by omitting one or more operations from a valid sequence

of operations is also valid.

Definition 3.3.1 *A property represented by an automaton $\mathcal{F}$ is said to be **omission-closed** when for all sequences $\alpha, \beta, \gamma \in \Sigma^*$, $Valid_{\mathcal{F}}(\alpha\beta\gamma) \Rightarrow Valid_{\mathcal{F}}(\alpha\gamma)$.*

The following theorem presents alternative characterizations of omission-closed properties.

Theorem 3.3.2 *Given an automaton $\mathcal{F}$, the following are all equivalent, where all sequences are elements of Σ^*:*

(a) For all sequences α, β, γ, $Valid_{\mathcal{F}}(\alpha\beta\gamma) \Rightarrow Valid_{\mathcal{F}}(\alpha\gamma)$.

(b) If ω_1 is a subsequence of ω_2, then $Valid_{\mathcal{F}}(\omega_2) \Rightarrow Valid_{\mathcal{F}}(\omega_1)$.

(c) There exists a finite set of forbidden subsequences *$\xi_1, \xi_2, \ldots, \xi_k$ such that a sequence α is in $Invalid_{\mathcal{F}}$ iff α contains some ξ_i as a subsequence.*

Proof: The equivalence of (a) and (b) is straightforward. As for, (c), consider the forbidden subsequences ξ_i corresponding to the *acyclic* paths in the automaton $\mathcal{F}$ from the initial state to the error state. Any sequence containing some ξ_i is invalid (from (b)), and it is clear that any invalid sequence must contain an acyclic path from the initial state to the error state as a subsequence. (For example, the forbidden subsequences for the automaton in Fig. 3.3 are ξ_1 = `close`;`read` and ξ_2 = `close`;`close`.) The result follows.

Example 3.3.3 *Consider the automaton $\mathcal{F}_{3.3}$ of Fig. 3.3. For this automaton, the sequence* `read`;`read`;`close` *is in $Valid_{\mathcal{F}_{3.3}}$, and so is the sequence* `read`;`close` *obtained by dropping the intermediate* `read` *operation. Moreover, for any valid sequence* `read`*;`close`*, dropping any subsequence of* `read`*s, or dropping the* `close` *yields a valid sequence.*

For $\mathcal{F}_{3.3}$, it is sufficient to consider the forbidden subsequences ξ_1 = `close`;`read` and ξ_2 = `close`;`close`. Each sequence α containing ξ_1 or ξ_2 as a subsequence is in $Invalid_{\mathcal{F}_{3.3}}$, and each sequence in $Invalid_{\mathcal{F}_{3.3}}$ contains ξ_1 or ξ_2 as a subsequence.

Background: Distributive Predicate Abstractions

The analysis we present will utilize a *predicate* abstraction that tracks the values of a set of predicates P defined on the concrete program-state. (We will use the term *program-state* to denote the state of the whole program in the concrete semantics, to distinguish it from a *state in an FSA specifying a property*.) For efficiency reasons, we will utilize an *independent attribute analysis* [80], an analysis that does not

maintain the correlation between different predicate values. Specifically, the set of concrete program-states arising at a program point will be abstracted by a value in $P \rightarrow \{false, maybe\}$. We now summarize the conditions under which an *independent attribute analysis* can be used for a predicate abstraction without losing precision. Given a predicate φ and a statement $\mathtt{St}$, we denote by $\mathrm{WP}(\mathtt{St}, \varphi)$ the weakest precondition of φ with respect to $\mathtt{St}$ [33].

Definition 3.3.4 *Given a finite set of predicates Base, we say that a finite set of predicates $\mathcal{P} = \{P_1, \ldots, P_k\}$ is a **distributive WP-closure** of Base when Base $\subseteq \mathcal{P}$ and for each predicate $P_i \in \mathcal{P}$, and for each statement $\mathtt{St}$, $\mathrm{WP}(\mathtt{St}, P_i) = P_{j_1} \vee \ldots \vee P_{j_m}$, where $P_{j_1}, \ldots, P_{j_m} \in \mathcal{P}$. We also say that the set of predicates $\mathcal{P}$ is* distributively WP-closed.

Theorem 3.3.5 *Given a distributively WP-closed set of predicates $\mathcal{P}$ for a program* Pgm, *precise analysis (i.e., determining for every program point and every predicate in $\mathcal{P}$ whether there exists a path to the program point causing the predicate to be* true*) is possible in time $O(|\mathcal{P}||\mathrm{Pgm}|)$.*

Proof: Straightforward. E.g., the problem can be reduced to a reachability problem over a graph of size $O(|\mathcal{P}||Pgm|)$, as in the IFDS framework of [85]. We note that the analysis can also identify paths that will cause a given predicate to become true at a given point when such a path exists.

A Polynomial Algorithm

We use a designated predicate *Error* that is *true* in a program-state if and only if the program-state contains an object in the error state *err*. We will now show that for omission-closed properties, a distributive WP closure of polynomial size can be constructed for $\{Error\}$. In general, a distributive WP closure for $\{Error\}$ needs to include predicates that refer to aliasing relationships among variables *as well as* the state of the objects pointed to by the variables. This motivates the following definition of a family of predicates.

Definition 3.3.6 *We write $In_\sigma(\mathsf{x})$ to denote the fact that the object pointed to by the variable x is in state $\sigma \in Q$. Given any $S \subseteq Q$, we use the shorthand $In_S(\mathsf{x}) \triangleq \bigvee_{\sigma \in S} In_\sigma(\mathsf{x})$ to denote that the object pointed to by the variable x is in one of the states in S.*

Definition 3.3.7 *Let A be a non-empty set of variables (in a given program) , $S \subseteq Q$ a set of states in $\mathcal{F}$. We use the predicate $\langle A, S\rangle$ to mean that all variables in A have the same value (are aliases), and the object referred to by variables in A is in one of the states in S. Formally,*

$$\langle A, S\rangle \triangleq \bigwedge_{\mathsf{x} \in A, \mathsf{y} \in A} (\mathsf{y} = \mathsf{x}) \wedge \bigwedge_{\mathsf{x} \in A} In_S(\mathsf{x})$$

```
V_F← = ∅; E_F← = ∅; workSet = {{err}};
while workSet ≠ ∅ {
  select and remove S from workSet;
  for each operation op ∈ Σ {
    P = δ←(S, op);
    if P ∉ V_F← { V_F← = V_F← ∪ {P}; workSet = workSet ∪ {P}; }
    E_F← = E_F← ∪ {P → S};
  }
}
```

Figure 3.4: Backwards exploration of the property automaton.

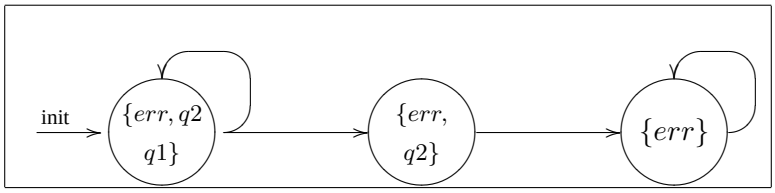


Figure 3.5: The graph constructed by backward exploration of the automaton of Fig. 3.3.

The number of predicates of the form $\langle A, S\rangle$ is exponential in the number of program variables. However, not all predicates of this form are *relevant*, i.e. need to be in a distributive WP closure for $\{Error\}$. The key to obtaining a polynomial size distributive WP closure for $\{Error\}$ is to bound the size of the set A, for any relevant predicate $\langle A, S\rangle$, by a constant. We will do this in two steps. First, we will show that a predicate $\langle A, S\rangle$ is relevant only for certain $S \subseteq Q$. Then, we will show that for each such set S, the predicate $\langle A, S\rangle$ is only relevant for A of cardinality less than a specific constant.

We first present an algorithm for determining which $S \subseteq Q$ are relevant for verification. The algorithm shown in Fig. 3.4 is based on a backward traversal of the finite state automaton. The algorithm constructs a graph $\overleftarrow{\mathcal{F}} = (V_{\overleftarrow{\mathcal{F}}}, E_{\overleftarrow{\mathcal{F}}})$, where each vertex is a subset of Q, and an edge $P \rightarrow S$ denotes that P is a pre-image of S for the transition function δ (see below).

Definition 3.3.8 *Let $\overleftarrow{\delta}$ denote the **reverse transition relation** of $\mathcal{F}$, i.e., given a state $q \in Q$, an operation $a \in \Sigma$, and a set of states $S \subseteq Q$, $\overleftarrow{\delta}(q, a) \triangleq \{q' \in Q | \delta(q', a) = q\}$, and $\overleftarrow{\delta}(S, a) \triangleq \bigcup_{q \in S} \overleftarrow{\delta}(q, a)$. For $S_1, S_2 \subseteq Q$, S_2 is said to be a* pre-image *of S_1 if $\exists a \in \Sigma. \overleftarrow{\delta}(S_1, a) = S_2$.*

Fig. 3.5 illustrates the graph constructed by backward exploration of the `read`*;`close` automaton shown in Fig. 3.3. We now establish a result about the graph $\overleftarrow{\mathcal{F}}$.

Stmt	WP(Stmt, $\langle A, S\rangle$)	
x := y	$\langle A[\mathsf{x} \mapsto \mathsf{y}], S\rangle$	
x := new ()	$\langle A, S\rangle$	if $\mathsf{x} \notin A$
	false	if $\mathsf{x} \in A \wedge A \neq \{\mathsf{x}\}$
	true	if $A = \{\mathsf{x}\} \wedge \mathit{init} \in S$
	false	if $A = \{\mathsf{x}\} \wedge \mathit{init} \notin S$
x.op()	$\langle A, S\rangle$	if $\overleftarrow{\delta}(S, \mathsf{op}) = S$
	$\langle A \cup \{x\}, \overleftarrow{\delta}(S, op)\rangle \ \vee \ \langle A, S\rangle$	if $\overleftarrow{\delta}(S, op) \supset S$
At program entry	*true*	if $\lvert A\rvert = 1 \wedge \mathit{init} \in S$
	false	if $\lvert A\rvert \neq 1 \vee \mathit{init} \notin S$

Figure 3.6: WP equations for predicates of the form $\langle A, S\rangle$. We denote by $A[x \mapsto y]$ the set obtained by replacing any occurrence of x in A by y.

Theorem 3.3.9 *If $\mathcal{F}$ represents an omission-closed property, then for any $S \in V_{\overleftarrow{\mathcal{F}}}$, and any operation $a \in \Sigma$, $\overleftarrow{\delta}(S, a) \supseteq S$. Further, the graph $\overleftarrow{\mathcal{F}}$ is acyclic except for self-loops.*

Proof: For any $S \in V_{\overleftarrow{\mathcal{F}}}$ there exists a sequence of operations ξ such that S is the set of all states in which ξ is invalid (by construction). Now, $\overleftarrow{\delta}(S, a)$ is the set of all states in which $a\xi$ is invalid. Since $\mathcal{F}$ is omission-closed, $\overleftarrow{\delta}(S, a) \supseteq S$. Since any predecessor P of S must be a superset of S, it follows immediately that any cycle in the graph $\overleftarrow{\mathcal{F}}$ must be a self-loop.

Fig. 3.6 and Fig. 3.7 present weakest-precondition equations for predicates of the form $\langle A, S\rangle$ and the special predicate *Error*. From these equations, we can determine which predicates are relevant for verification. The equations reveal two things. First, they show that it is sufficient to restrict our attention to predicates of the form $\langle A, S\rangle$ where $S \in V_{\overleftarrow{\mathcal{F}}}$. Second, they show that a predicate $\langle A, P\rangle$ is relevant only if there is a relevant predicate $\langle B, S\rangle$ where S is a proper successor of P in the graph $\overleftarrow{\mathcal{F}}$ and B has cardinality at least $|A| - 1$. In other words, we need only consider predicates of the form $\langle A, P\rangle$ where the cardinality of A is less than or equal to the length of the longest acyclic path from P to $\{err\}$ in $\overleftarrow{\mathcal{F}}$.

Definition 3.3.10 *For any $S \in V_{\overleftarrow{\mathcal{F}}}$, define $dist(S)$ to be the number of edges in the longest acyclic path from S to* {err} *in $\overleftarrow{\mathcal{F}}$. Given a program with a set of variables* Vars*, we define a set of predicates* $\mathcal{P} = \{\langle A, S\rangle | S \in V_{\overleftarrow{\mathcal{F}}}, A \subseteq \text{Vars}, |A| \leq dist(S)\} \cup \{\mathit{Error}\}$.

Theorem 3.3.11 *The set* $\mathcal{P} \cup \{\text{true}, \text{false}\}$ *is a distributively WP-closed set of predicates for* $\{\mathit{Error}\}$.

Proof: Follows from the above discussion.

Theorem 3.3.12 *If $\mathcal{F}$ is omission-closed, then* $\text{SV}_{\mathcal{F}}$ *is in P.*

`Stmt`	WP(`Stmt`, *Error*)
`x := y`	*Error*
`x := new ()`	*Error*
`x.op()`	*Error* if $\overleftarrow{\delta}(\{err\}, op) = \{err\}$ $\langle\{x\}, \overleftarrow{\delta}(\{err\}, op)\rangle \vee$ *Error* if $\overleftarrow{\delta}(\{err\}, op) \supset \{err\}$
At program entry	*false*

Figure 3.7: WP equations for the predicate *Error*.

Proof: Immediate from Theorem 3.3.11 and Theorem 3.3.5. Note that the cardinality of $\mathcal{P}$ is $O(|Vars|^k)$, where *Vars* is the set of all variables in the program and k is the length of the longest acyclic path in $\overleftarrow{\mathcal{F}}$. (Note, from Theorem 3.3.9, that k is also bounded by the number of states in $\mathcal{F}$.)

Example 3.3.13 *Consider the property $read^*close$ represented by the automaton of Fig. 3.3. The graph $\overleftarrow{\mathcal{F}}$ for this automaton is shown in Fig. 3.5. The derivation for this property is as follows*[2]*:*

$$\text{WP}(\texttt{x.read()}, \mathit{Error}) = \langle\{x\}, \{\text{err}, q_2\}\rangle \vee \mathit{Error}$$
$$\text{WP}(\texttt{x.close()}, \mathit{Error}) = \langle\{x\}, \{\text{err}, q_2\}\rangle \vee \mathit{Error}$$
$$\text{WP}(\texttt{y.close()}, \langle\{x\}, \{\text{err}, q_2\}\rangle) = \langle\{x, y\}, \{\text{err}, q_2, q_1\}\rangle \vee \langle\{x\}, \{\text{err}, q_2\}\rangle$$
$$\text{WP}(\texttt{w.read()}, \langle\{x, y\}, \{\text{err}, q_2, q_1\}\rangle) = \langle\{x, y\}, \{\text{err}, q_2, q_1\}\rangle$$

Thus, `read`*;`close` *verification can be done in time* $O(|\text{Vars}|^2|\text{Pgm}|)$.

Discussion

A logical formula can usually be simplified into a number of equivalent forms. Hence, a weakest-precondition can often be expressed in many ways. The form we chose to use in expressing weakest-preconditions above is critical to deriving a polynomial-time verification algorithm. As an example, consider the `read`*;`close` example. The following is an alternative, correct, weakest-precondition equation, which says that an object in the *err* state is possible after `x.close()` iff either `x` points to an object in state q_2 or an object exists in the *err* state before the statement:

$$\text{WP}(\texttt{x.close()}, \mathit{Error}) = \langle\{x\}, \{q_2\}\rangle \vee \mathit{Error}. \tag{3.1}$$

The actual formulation we used

$$\text{WP}(\texttt{x.close()}, \mathit{Error}) = \langle\{x\}, \{err, q_2\}\rangle \vee \mathit{Error} \tag{3.2}$$

[2]Note that the variables `x`, `y`, and `w` used in the derivation process are free variables and not variables of a specific program.

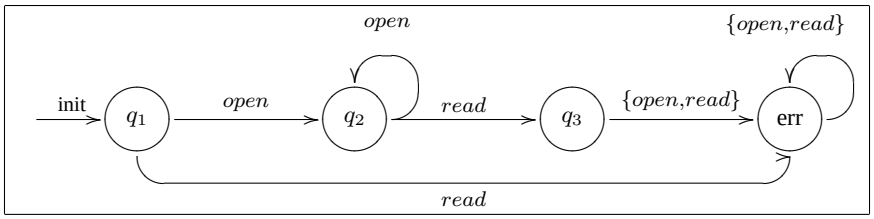


Figure 3.8: An automaton for the property $\texttt{open}^+;\texttt{read}$.

actually contains some redundancy. In particular, $\langle\{\texttt{x}\},\{err,q_2\}\rangle$ is equivalent to $\langle\{\texttt{x}\},\{err\}\rangle \vee \langle\{\texttt{x}\},\{q_2\}\rangle$. But the disjunct $\langle\{\texttt{x}\},\{err\}\rangle$ is redundant because it implies *Error*, another disjunct in our formula.

However, equation 3.2 is preferable to equation 3.1. In particular, we have seen that we can determine in polynomial time if $\langle\{\texttt{x}\},\{err,q_2\}\rangle$ is possible at any program point. However, one can show that determining if $\langle\{\texttt{x}\},\{q_2\}\rangle$ is possible at a program point is PSPACE-hard, adapting the proof we present in Section 3.4. Thus, unless PSPACE = P, a distributively WP-closed set containing $\langle\{\texttt{x}\},\{q_2\}\rangle$ of polynomial size *does not exist*! Note that the set $\{q_2\}$ has a pre-image (namely $\overleftarrow{\delta}(\{q_2\},\texttt{close}) = \{q_1\}$) that is not a superset of $\{q_2\}$, thus not satisfying the requirements of Theorem 3.3.9. This is why the proof used for omission-closed properties cannot be used for this predicate.

3.4 Repeatable Enabling Sequence Properties

In this section we show that verification of Repeatable Enabling Sequence properties (see Definition 3.4.1) is NP-complete for acyclic programs and PSPACE-complete in general.

Definition 3.4.1 (Repeatable Enabling Sequence Properties) *We say that a property represented by an automaton $\mathcal{F}$ is a repeatable enabling sequence property if there exist sequences of operations α, β and γ such that the set of sequences $\alpha\beta^+\gamma$ are all valid but the sequence $\alpha\gamma$ is invalid. (The sequence β may be thought of as a repeatable sequence that enables γ.)*

For example, the property $\texttt{open}^+;\texttt{read}$ (see Figure 3.8) which requires that a `read` be preceded by one or more `open` operations is a repeatable enabling sequence property. (The more natural property $\texttt{open}^+;\texttt{read}^*$ is also a repeatable enabling sequence property, but we use $\texttt{open}^+;\texttt{read}$ as the running example to contrast it with the omission-closed property $\texttt{read}^*;\texttt{close}$.) We show that verification of repeatable enabling sequence properties is PSPACE-complete by reduction from the *simultaneously false* problem (see [74], [40]).

Definition 3.4.2 (Simultaneously False Problem) *Given a program P with an initial assignment of values (0 or 1) to a set $x_1, x_2, \ldots, x_n$ of boolean variables, where the program P contains only assignments (of constants or variables), conditionals or unconditional jumps, a* simultaneously false *problem*

for P is a problem of the form: is there an execution path from the entry point of P to a program point p such that $x_1 = 0, x_2 = 0, \ldots x_k = 0$ when control reaches p ?

Lemma 3.4.3 *(1) The simultaneously false problem for acyclic programs is NP-complete. (2) The simultaneously false problem for arbitrary programs is PSPACE-complete.*

Proof: The binary simultaneous value problem can be easily reduced to the simultaneously false problem by following the construction used in the proof of Theorem 3.6 in Muth and Debray [74]. The idea is to transform a program P into a program P' such that every variable x_i in P corresponds to two variables X_i and $\overline{X_i}$, every assignment $x_i = 0$ is converted to $X_i = 0; \overline{X_i} = 1$, every assignment $x_i = 1$ is converted to $X_i = 1; \overline{X_i} = 0$, and every assignment $x_i = x_j$ is converted into $X_i = X_j; \overline{X_i} = \overline{X_j}$. Consider the simultaneous value problem $x_1 = c_1, x_2 = c_2, \ldots, x_k = c_k$ for P. It can be easily shown that the simultaneously false problem for P' obtained by replacing every conjunct $x_i = 0$ with $X_i = 0$ and $x_i = 1$ with $\overline{X_i} = 0$ is equivalent. Thus, the simultaneously false problem is also NP-complete and PSPACE-complete for acyclic and arbitrary programs respectively.

Let $\mathcal{F}$ be an automaton representing a repeatable enabling sequence property. We show that $\text{SV}_{\mathcal{F}}$ is PSPACE-hard by reduction from the simultaneously false problem. If α, β, γ are such that sequences $\alpha\beta^+\gamma$ are valid and sequence $\alpha\gamma$ is invalid, then β and γ must be non-empty (although α may be empty). Given an instance of the simultaneously false problem $x_1 = 0, x_2 = 0, \ldots, x_k = 0$ at program point p in a program P, we construct a program P' as follows. First, we create two objects `Zero` and `One` which support methods corresponding to the sequences α, β, and γ. Next, we copy program P into P' replacing every assignment of the form $\texttt{x}_\texttt{i} = \texttt{0}$ by $\texttt{x}_\texttt{i} = \texttt{Zero}$ and $\texttt{x}_\texttt{i} = \texttt{1}$ by $\texttt{x}_\texttt{i} = \texttt{One}$ respectively. Then, at program point p, we insert the statement `if` $(?)$ `goto` $\texttt{p}_1$. Let the sequence α be $a_1, a_2, \ldots, a_l$, let β be $b_1, b_2, \ldots b_m$, and let γ be $c_1, c_2, \ldots c_n$. We insert the following sequence of statements at the end.

```
            goto exit;
p1 :        Zero.a1(); Zero.a2(); ...; Zero.al();
            One.a1(); One.a2(); ...; One.al();
            x1.b1(); x1.b2(); ...; x1.bm();
            x2.b1(); x2.b2(); ...; x2.bm();
            ...
            xk.b1(); xk.b2(); ...; xk.bm();
            One.c1(); One.c2(); ...; One.cn();
exit :
```

Note that control can reach program point $\texttt{p}_1$ only through the conditional branch statement `if` $(?)$ `goto` $\texttt{p}_1$ (because of the statement `goto exit;` just before $\texttt{p}_1$).

Lemma 3.4.4 *Assuming that the sequences of operations β and γ are non-empty, the simultaneously false problem $x_1 = 0, x_2 = 0, \ldots x_k = 0$ at program point p in P returns true if and only if program P' violates the property represented by $\mathcal{F}$.*

Proof: Program P' creates only two objects $Zero$ and One. Note that the only sequence of operations performed on $Zero$ is $\alpha\beta^i$ where i is the number of variables in $x_1, x_2, \ldots, x_k$ that are aliased to $Zero$ at program point p. Thus, no illegal operation is ever performed on $Zero$. The only sequence of operations performed on One is $\alpha\beta^j\gamma$ where j is the number of variables in $x_1, x_2, \ldots, x_k$ that are aliased to One at program point p. This sequence is invalid iff j can be 0. In other words, P' violates the property represented by $\mathcal{F}$ iff the simultaneously false problem $x_1 = 0, x_2 = 0, \ldots x_k = 0$ at program point p in P returns true.

The above lemma shows the hardness of typestate verification for repeatable enabling sequence properties. We now establish a straightforward completeness result.

Lemma 3.4.5 *For any automaton $\mathcal{F}$, $\text{SV}_\mathcal{F}$ is in NP for acyclic programs and in PSPACE for arbitrary programs.*

Proof: $\text{SV}_\mathcal{F}$ is in NP for acyclic programs since we can non-deterministically choose a path through the program and check to see if any object reaches the error state during execution along that path. To show that $\text{SV}_\mathcal{F}$ for an arbitrary program P is in PSPACE, we construct a non-deterministic multi-tape polynomial-space-bounded Turing Machine M to solve the problem. M simulates input program P, non-deterministically choosing the branch to take at branch points. Let us refer to objects pointed to by the variables in P as *live* objects. M keeps track of which variables point to which (live) objects, and tracks the finite-state of each live object. The space needed to maintain this information is trivially bounded by a polynomial in the size of program P. If any of the relevant objects goes into the error state during simulation, M halts and signals the possibility of an error. Conversely, if there is a path that causes one of the objects to go into the error state, then M can guess this path and will halt signalling the error.

Theorem 3.4.6 *Consider a repeatable enabling sequence property represented by an automaton $\mathcal{F}$. $\text{SV}_\mathcal{F}$ is NP-complete for acyclic programs and PSPACE-complete for arbitrary (cyclic) programs.*

Proof: The proofs of NP-hardness and PSPACE-hardness of acyclic and arbitrary programs resepectively follows from Lemmas 3.4.3 and 3.4.4 respectively. Lemma 3.4.5 shows that the problem of shallow verification for all safety properties represented by an automaton is in NP for acyclic programs and in PSPACE for arbitrary programs.

Theorem 3.4.6 shows that verification of repeatable enabling sequence properties is difficult even for shallow programs. In fact, the situation is worse. We now show that even the shortest error paths may be of exponential size in the worst case.

Definition 3.4.7 (Error Path) *Let $\mathcal{F}$ be an automaton representing a property to be verified. We say that a (possibly cyclic) path in the control flow graph of P from the entry vertex to some vertex v is an* error path *if symbolic execution of the program along this path (ignoring the conditionals) exhibits a violation of the property associated with $\mathcal{F}$. The program P is said to be* erroneous *if there exists an error path in P. An integer-valued function f is said to be a bound on the shortest error path length if every erroneous program for size n is guaranteed to have an error path of length $f(n)$ or less.*

Definition 3.4.8 (Loop Unrolling) *Consider the control-flow-graph $G_P = (V_P, E_P)$ of program P. Let $G'_P = (V_P, E'_P)$ denote the acyclic graph obtained from G_P by removing all back-edges. We define Unroll(G_P, n) to be the acyclic graph obtained by making $n+1$ copies of G'_P (called $G'_P(1), G'_P(2), \ldots$ $G'_P(n+1)$ respectively), and for every back-edge (u, v) in G_P, adding an edge from vertex u in $G'_P(i)$ to vertex v in $G'_P(i+1)$ for all i from 1 to v. More formally Unroll$(G_P, n) = (V^*, E^*)$ where*

$$\begin{aligned} V^* = &\ \{\ (v,i) \mid\ v \in V_P,\ 1 \le i \le n+1\ \} \\ E^* = &\ \{\ [(u,i),(v,i)] \mid\ [u,v] \in E'_P,\ 1 \le i \le n+1\ \} \cup \\ &\ \{[(u,i),(v,i+1)] \mid [u,v] \in E_P - E'_P, 1 \le i \le n\ \} \end{aligned}$$

It is easy to verify that *Unroll*(G_P, v) is acyclic and contains every path of length v or less in G_P.

Theorem 3.4.9 *If NP $\neq$ PSPACE, then there does not exist a polynomial bound on the shortest error path length for repeatable enabling sequence properties.*

Proof: Let $\mathcal{F}$ be the finite state automaton associated with the repeatable enabling sequence property. From Theorem 3.4.6 it follows that verification of $\mathcal{F}$ for acyclic programs is in NP and for arbitrary (cyclic) programs is PSPACE-hard. We prove Theorem 3.4.9 by showing that if there is a polynomial bound on the shortest error path, then the verification problem for cyclic programs can be polynomial-time reduced to the verification problem for acyclic programs, which would imply that NP = PSPACE.

Let $p(n)$ denote a polynomial bound on the size of the shortest error path where n denotes the size of the program. Given an arbitrary program P with control flow graph G_P, we construct the acyclic program *Unroll*$(G_P, p(n))$ which is acyclic and contains all paths of length $p(n)$ or less in G_P. The size of *Unroll*$(G_P, p(n))$ and the time taken to construct it are both polynomial in n. Thus, the problem of verification of G_P is polynomially reduced to the problem of verifying *Unroll*$(G_P, p(n))$, which is a contradiction.

Theorem 3.4.9 suggests that it may not be possible to find short counterexample paths exhibiting the violation of properties like `open`$^+$;`read`. This is important to know because many approaches to verification (e.g., [6]) are inherently associated with the generation of a counterexample path that exhibits the violation of the property of interest. Theorem 3.4.9 suggests the possibility that even the shortest error path in the program may be of size exponential in the size of the program.

3.5 Verification by counting

We have now seen that verification is intractable for repeatable enabling sequence properties and polynomial for omission-closed properties. Unfortunately, there are properties that fall into neither class. A simple example is the `open`; `read` property. Note that `open`; `read` is similar to `open`$^+$; `read` in that it requires that an object be opened before it can be read, but it differs from it in that an object cannot be opened multiple times. Does this make verification any easier?

3.5.1 The Intuition

The requirement that an object cannot be opened multiple times is a forbidden subsequence problem (where `open`; `open` is the forbidden subsequence) (see Theorem 3.3.2(c)). It follows that we can verify if the given program may open an object multiple times in polynomial time. Thus, `open`; `read` verification is polynomial-time equivalent to `open`$^+$; `read` verification of a program *guaranteed not to open any object more than once*. We will now show that, at least for acyclic programs, this added restriction (that an object cannot be opened multiple times) does make polynomial-time verification possible.

Let us begin by considering why `read`*; `close` verification is easy while `open`$^+$; `read` verification is not. Consider the following code fragment:

```
...; p1.open(); ...; pk.open(); ...; q.read();
```

The `open`$^+$; `read` property will be violated if there is an execution path such that the value of `q` at the `read` statement is different from the values of *each* $\mathtt{p}_i$ at the corresponding `open` statements (assuming there are no `open` statements in the program other than those shown above). Determining if certain relationships can *simultaneously* exist among a potentially unbounded number of program variables is difficult.

In contrast, consider the following code fragment:

```
...; p1.close(); ...; pk.close(); ...; q.read();
```

The `read`*; `close` property will be violated here if there is an execution path such that the value of `q` at the `read` statement is equal to the value of *some* $\mathtt{p}_i$ at the corresponding `close` statement. In other words, this requires *independent* answers to `k` different questions, each about the value of only *two* program variables. This turns out to be easy.

Let us now turn back to the earlier example above.

```
...; p1.open(); ...; pk.open(); ...; q.read();
```

If we now know that no object is opened twice, how can we exploit this for `open`$^+$; `read` (i.e., `open`; `read`) verification? For any given `i`, we know that it is easy to determine if the `q.read()`

statement may read the same object that is opened by the $\texttt{p}_i\texttt{.open()}$ statement. Imagine that we can *count* the number of execution paths, n_i, along which this can happen, for each `i`. Adding up all the n_i would tell us how many times (i.e., along how many execution paths) the `q.read()` statement is a *valid* operation[3]. If this number does not equal the number of execution paths to the `q.read()` statement, then *there must be an execution path along which* `q.read()` *will read an unopened object*! Such indirect reasoning based on counting is the basis for the algorithm presented in [41].

3.6 Programs with Width-Limited Aliasing

In Section 3.4 we saw that, unless P = NP, verification of repeatable enabling sequence properties will require exponential time *in the worst-case*. Is it, however, possible to design verification algorithms that are efficient *in practice*, e.g., by exploiting properties of programs that arise in practice? For example, one seldom sees programs in which a very large number of variables point to the same object at a program point. Let us say that a program has a maximum *aliasing width* of k if there is no execution path in the program that will produce an object pointed to by more than k different variables. In this section, we look at the complexity of typestate verification for programs where the maximum aliasing width is bounded by a constant.

3.6.1 Polynomial-Time Verification for Shallow Programs with Width-Limited Aliasing

In this section we present a verification algorithm motivated by the observation that the aliasing width of programs tends to be small in practice. The algorithm runs in time $O(|Pgm|^{k+1})$, where $|Pgm|$ is the size of the program and k is the maximum aliasing width of the program: Unlike the polynomial solutions of previous sections, the algorithm presented here works for any typestate property.

We note that naive verification algorithms do not achieve the above complexity, i.e. they may take exponential time even for programs with a maximum aliasing width of 2. In particular, consider the obvious abstraction where the program-state is represented by a partition of the program variables into equivalence classes (of variables that are aliased to each other), with a finite state associated with each equivalence class. The number of such program-states that can arise at a program point is exponential in the number of program variables even for programs with a maximum aliasing width of 2.

Our algorithm uses predicates of the form $[A, S]$ defined below.

Definition 3.6.1 *Let* $A \subseteq$ Vars *be a non-empty set of program variables, and* $S \subseteq \mathcal{Q}$ *a set of states of* $\mathcal{F}$*.*

$$[A, S] = \bigwedge_{x \in A, y \in A} (y = x) \ \wedge \bigwedge_{x \in A, z \in \text{Vars} \setminus A} (z \neq x) \ \wedge \bigwedge_{x \in A} In_S(x) \ )$$

[3] This is where we exploit the fact that no object is opened twice. Otherwise, adding up n_i will end up counting some paths multiple times.

`Statement`	*flow*(`Statement`)($[A, \sigma]$)	
`x := y`	$\{[A \cup \{x\}, \sigma]\}$	if $y \in A$
	$\{[A \setminus \{x\}, \sigma]\}$	if $y \notin A$
`x := new()`	$\{[\{x\}, \mathit{init}], [A \setminus \{x\}, \sigma]\}$	if $x \in A$
	$\{[A, \sigma]\}$	if $x \notin A$
`x.op()`	$\{[A, \delta(\sigma, op)]\}$	if $x \in A$
	$\{[A, \sigma]\}$	if $x \notin A$

Figure 3.9: *flow* equations for predicates of the form $[A, \sigma]$.

When S contains a single state $\sigma \in \mathcal{Q}$, we write $[A, \sigma]$, rather than $[A, \{\sigma\}]$.

Intuitively, a predicate $[A, S]$ means that all variables in A have the same value (are aliases), every variable not in A has a different value from the variables in A, and the object referred to by variables in A is in one of the state of S. The difference between $[A, S]$ and $\langle A, S\rangle$ (Definition 3.3.7) is noteworthy. The non-aliasing conditions are implicitly represented in $[A, S]$ by assuming that every variable not in A has a different value from the variables in A, whereas in $\langle A, S\rangle$, the variables not in A may or may not be aliased to the variables in A.

Fig. 3.10 presents our verification algorithm that computes, for all program points, the set of predicates of the form $[A, \sigma]$ that may-be-true at the program point. (A predicate p is said to be may-be-true at a program point u iff there exists a path to u such that execution along that path will cause p to become true.) The algorithm is based on a standard iterative collecting interpretation algorithm. The function *flow*(`St`)(φ), defined in Fig. 3.9, identifies the set of predicates that may-be-true after statement `St` given a predicate φ that may-be-true before statement `St`. For any program point l, $Succ(l)$ denotes the successors of l.

Theorem 3.6.2 *The algorithm of Fig. 3.10* precisely *computes the set of predicates $[A, S]$ that may hold at any program point in time $O((\sum_{1 \le i \le k} \binom{n}{i}) * |\text{Pgm}|) = O(n^k * |\text{Pgm}|)$ where k is the maximum number of variables aliased to each other at any point in the program* Pgm, *and $n = |\text{Vars}|$ is the number of program variables.*

Proof: It can be shown that (a) $\cup_{\varphi \in P}$*flow*(`St`)(φ) computes a precise abstract transfer function for statement `St` with respect to the set of predicates P, and that (b) this is a distributive function. It directly follows from these facts that the algorithm computes the precise solution.

We now establish the complexity of the algorithm. Assume that the maximal size of an alias-set occurring in the program is k. The algorithm may generate predicates of the form $[A, S]$ for all subsets of any size up to k of program variables *Vars*. The number of predicates that may have a

```
workList = {}
for each program point l
  results(l) = {}
for each program variable x_i
  add (entry, [x_i, {init}]) to workList
while workList ≠ ∅ {
  remove (l, ψ) from workList
  for each ψ' ∈ flow(stmt_l)(ψ) {
    for l' ∈ Succ(l) {
      if ψ' ∉ results(l') {
        results(l') = results(l') ∪ {ψ'}
        add (l', ψ') to workList
      }
    }
  }
}
```

Figure 3.10: An iterative algorithm using predicates of the form $[A, S]$.

true value in a program point is therefore $O(\sum_{1 \leq i \leq k} \binom{n}{i})$ where $n = |\mathit{Vars}|$ (we treat the number of FSM states as a constant). The complexity of the chaotic iteration algorithm of Fig. 3.10 is therefore $O((\sum_{1 \leq i \leq k} \binom{n}{i}) * |\mathit{Pgm}|)$. The expression is also bounded by $O(n^k * |\mathit{Pgm}|)$. The above assumes that the step of computing $\mathit{flow}(\texttt{stmt}_l)(\psi)$ takes constant time.

Though the worst-case complexity of the algorithm is exponential, the exponential factor k is expected to be a small constant for typical programs, since the number of pointers simultaneously pointing to the same object is expected to be small (and significantly smaller than $|\mathit{Vars}|$).

Note that using the set of predicates defined in Definition 3.6.1 is not sufficient to achieve the desired complexity. The style of "forward propagation" used by our algorithm is also essential, as it ensures that the cost of analysis is proportional to the number of predicates that may-be-true (rather than the number of total predicates, as is the case with alternative analysis techniques).

3.6.2 Width-Limited Aliasing in Non-Shallow Programs

We have now seen that typestate verification can be done efficiently for programs where the aliasing is bounded in certain ways. Specifically, the results of the previous subsection show that for shallow programs, typestate verification can be done in polynomial time if the aliasing width is assumed to be bounded by a constant. A natural question is whether any such result holds true for non-shallow programs.

Recall that shallow programs are programs where the aliasing *depth* is restricted to be one: program variables may point to objects, but the program contains no variables that point to objects that contain pointers to objects.

Unfortunately, it turns out that typestate verification is hard for non-shallow programs even if aliasing width is bounded by a constant. It is known [61] that alias analysis is intractable for programs where the aliasing depth is two. We now show that the intractability result holds even if in addition the aliasing width is also restricted to three.

Theorem 3.6.3 *Alias analysis is NP-hard for programs with aliasing depth two and aliasing width three.*

Proof: The proof is by reduction from 3-SAT. Consider a 3-SAT formula $C_1 \wedge C_2 \cdots \wedge C_n$ over logical variables w_1 through w_m. We create a program with a type `T` and a second type `PT` consisting of a field `f` of type (pointer to) `T`. Corresponding to every clause C_i, the program consists of variables $\texttt{X}_\texttt{i}$, $\texttt{Y}_{\texttt{i,true}}$, and $\texttt{Y}_{\texttt{i,false}}$ of type (pointer to) `PT` initialized as follows:

```
Y_i,true = new PT(); Y_i,true.f = new T();
Y_i,false = new PT(); Y_i,false.f = new T();
X_i = Y_i,false
```

Both $\texttt{Y}_{\texttt{i,true}}$ and $\texttt{Y}_{\texttt{i,false}}$ are constants in the program.

After the initialization code, the program consists of one if-then-else statement for every logical variable w_i in the 3-SAT formula. The then-branch of this statement consists of an assignment statement $\texttt{X}_\texttt{i} = \texttt{Y}_{\texttt{i,true}}$ for every clause C_i that contains the literal w_i as one of its disjuncts. The else-branch of this statement consists of a similar assignment statement $\texttt{X}_\texttt{i} = \texttt{Y}_{\texttt{i,true}}$ for every clause C_i that contains the negated literal $\overline{w_i}$ as one of its disjuncts.

Thus, there exists a one-to-one correspondence between execution paths through the m if-then-else statements and possible truth assignments to the m logical variables, where we associate the then-branch of the i-th if-statement with an assignment of true to logical variable w_i. It should be clear that after execution through any path, $\texttt{X}_\texttt{i}$ points to the same object as $\texttt{Y}_{\texttt{i,true}}$ iff the corresponding truth assignment makes clause C_i to evaluate to true.

We now append the following code fragment:

```
S = new T();
Y1,true.f = S;
Y2,true.f = X1.f; Y1,true.f = new T();
Y3,true.f = X2.f; Y2,true.f = new T();
...
Yn,true.f = Xn−1.f; Yn−1,true.f = new T();
R = Yn,true.f;
```

Now, consider any execution path through the whole program that corresponds to a truth assignment that makes the entire formula true. Then, a pointer to the object created by the statement $\texttt{S} = \texttt{new T}();$ will be successively copied through every $\texttt{Y}_{\texttt{i,true}}\texttt{.f}$ and then finally to R, causing S and R to be aliased at the end of the program. Conversely, it can be verified that an execution path will cause S and R to be aliased to each other at the end of the program only if the path corresponds to a truth assignment that makes the given 3-SAT formula true.

Hence, R and S may alias each other at the end of the program iff the given 3-SAT formula is satisfiable.

Note that the program generated above has an aliasing width of three (i.e., no more than three pointers point to the same object at any point during program execution). In particular, the assignments $\texttt{Y}_{\texttt{i,true}}\texttt{.f = new T();}$ guarantee that no more than 3 pointers could point to S at any given time.

The following theorem is a straightforward consequence of the above result.

Theorem 3.6.4 *Typestate verification is NP-hard for programs with aliasing depth two and aliasing width three.*

3.7 Conclusion

In this chapter we have shown that verification of omission-closed properties is in P and that verification of repeatable enabling sequence properties is NP-complete for acyclic programs and PSPACE-complete in general. We have shown that verification of almost-omission-closed properties is in P for acyclic programs. However, many questions still remain open. E.g., we do not know if verification of almost-omission-closed properties is in P for cyclic programs. Moreover there are properties which do not lie in any of these classes. E.g., consider the property $\texttt{open};\texttt{read}^*$ which generalizes $\texttt{open};\texttt{read}$ by allowing any number of `read` operations. We can adapt the *counting* method of [41] to show that verification of $\texttt{open};\texttt{read}^*$ is in P for acyclic programs. However, we have not been able to formulate such a result for a general class of properties that includes $\texttt{open};\texttt{read}^*$. Finally, there are also other properties such as $(\texttt{lock};\texttt{unlock})^*$ (any number of alternating `lock` and `unlock` operations) for which we have neither been able to show a polynomial bound, nor an NP-hardness result.

On a more pragmatic note, we have presented a typestate verification algorithm, for arbitrary typestate properties, that we expect will perform well based on the reasonable assumption that programs tend to have small aliasing width. However, this algorithm is restricted to shallow programs. A natural question is how these ideas can be generalized to conduct verification for arbitrary programs. One of the primary intuitions behind our verification algorithm (for shallow programs) is that maintaining just the right correlation required between "analysis facts" can be the key to efficient and precise verification: maintaining no correlations (independent attribute analysis) can lead to imprecision, while maintaining all correlations (relational analysis) can lead to inefficiency. The techniques presented in Chapter 6 (and in [117]) show one way to exploit this intuition for verification of arbitrary (i.e. non-shallow) programs as well.

Chapter 4

Verifying Temporal Heap Properties Specified via Evolution Logic

This chapter addresses the problem of establishing temporal properties of programs written in languages, such as Java, that make extensive use of the heap to allocate—and deallocate—new objects and threads. Establishing liveness properties is a particularly hard challenge. One of the crucial obstacles is that heap locations have no static names and the number of heap locations is unbounded. The chapter presents a framework for the verification of Java-like programs. Unlike classical model checking, which uses propositional temporal logic, we use evolution temporal logic (ETL), a first-order temporal logic, to specify temporal properties of heap evolutions; this logic allows domain changes to be expressed, which permits allocation and deallocation to be modelled naturally.

In this chapter and in Chapter 5, we present two verification algorithms for ETL, based on two alternative semantics. In this chapter, we provide a *varying-domain semantics* in which the semantics of the program is considered to be a set of (infinite) traces in which each configuration may have its own domain. Then, in Chapter 5, we describe a *constant-domain semantics* in which all configurations along a trace share a single constant (infinite) domain. Using constant domain semantics allows us to naturally define a state-based semantics for ETL resulting in a more efficient verification algorithm.

Space by itself, and time by itself, are doomed to fade away into mere shadows,
and only a kind union of the two will preserve an independent reality.
–Albert Einstein.

4.1 Introduction

Modern programming languages, such as Java, make extensive use of the heap. The contents of the heap may evolve during program execution due to dynamic allocation and deallocation of objects. Moreover, in Java, threads are first-class objects that can be dynamically allocated. Statically reasoning about

temporal properties of such programs is quite challenging, because there are no *a priori* bounds on the number of allocated objects, or restrictions on the way the heap may evolve. In particular, proving liveness properties of such programs, e.g., that a thread is eventually created in response to each request made to a web server, can be quite a difficult task.

The contributions of this chapter can be summarized as follows:

- We introduce a first-order modal (temporal) logic [45, 42] that allows specifications of temporal properties of programs with dynamically evolving heaps to be stated in a natural manner.
- We develop an abstract interpretation [24] for verifying that a program satisfies such a specification.
- We implemented a prototype of the analysis using the TVLA system [64] and applied it to verify several temporal properties, including liveness properties of Java programs with evolving heaps.

We have used the framework to specify and verify the following:

Specify general heap-evolution properties: The framework has been used to specify, in a general manner, various properties of heap evolution, such as properties of garbage-collection algorithms.

Verify termination of sequential heap-manipulating programs: Termination is shown by providing a ranking function based on the set of items reachable from a variable iterating over the linked data structure. In particular, we have verified termination of all example programs from [35].

Verify temporal properties of concurrent heap-manipulating programs: We have used the framework to verify temporal properties of concurrent heap-manipulating programs — in particular, liveness properties, such as the absence of starvation in programs using mutual exclusion, and response properties [69]. We have applied this analysis to programs with an unbounded number of threads.

The remainder of this chapter is organized as follows: Section 4.2 gives an overview of the verification method and contrasts it with previous work. Section 4.3 introduces trace semantics based on first-order modal logic, and discusses how to state trace properties using the language of evolution logic. Section 4.4 defines an implementation of trace semantics via first-order logic. Section 4.5 shows how abstract traces are used to conservatively represent sets of concrete traces. Section 4.6 summarizes related work. Finally, Section 4.7 concludes the chapter.

4.2 Overview

4.2.1 A Temporal Logic Supporting Evolution

The specification language, *Evolution Temporal Logic* (ETL), is a first-order linear temporal logic that allows specifying properties of the way program execution causes dynamically allocated memory (“the heap”) to evolve.

It is natural to consider the concrete semantics of a program as the set of its execution traces [26, 107], where each trace is an infinite sequence of *worlds*. First-order logical structures provide a natural representation of worlds with an unbounded number of objects: an individual of the structure's domain (universe) corresponds to an anonymous, unique store location, and predicates represent properties of store locations. Such a representation allows properties of the heap contents to be maintained while abstracting away any information about the actual physical locations in the store.

This gives rise to traces in which worlds along the trace may have different domains. Such traces can be seen as models of a first-order modal logic with a varying-domain semantics [42]. This could be equivalently, but less naturally, modelled using constant-domain semantics.

This framework generalizes other specification methods that address dynamic allocation and deallocation of objects and threads. In particular, its descriptive power goes beyond Propositional LTL and finite-state machines (e.g., [17]).

Program properties can be verified by showing that they hold for all traces. Technically, this can be done by evaluating their first-order modal-logic formulae against all traces. We use a variant of Lewis' counterpart theory [66] to cast modal models (and formula evaluation) in terms of classical predicate logic with transitive closure (FO^{TC}) [23].

Program verification using the above concrete semantics is clearly non-computable in general. We therefore represent potentially infinite sets of infinite concrete traces by one abstract trace. Infinite parts of the concrete traces are folded into cycles of the abstract traces. Termination of the abstract interpretation on an arbitrary program is guaranteed by bounding the size of the abstract trace. Two abstractions are employed: (i) representing multiple concrete worlds by a single abstract world, and (ii) creating cycles when an abstract world reoccurs in the trace.

Because of these abstractions, we may fail to show the correctness of certain programs, even though they are correct. Fortunately, we can use reduction arguments and progress monitors as employed in other program-verification techniques (e.g., [58]).

As in finite-state model checking (e.g., [107]), we let the specification formula affect the abstraction by making sure that abstract traces that fulfill the formula are distinguished from the ones that do not. However, our abstraction does not fold the history of the trace into a single state. This idea of using the specification to affect the precision of the analysis was not used in [91, 115], which only handle safety properties.

4.2.2 Overview of the Verification Procedure

First, the property φ is specified in ETL. The formula is then translated in a straightforward manner into an FO^{TC} logical formula, $(\varphi)^{\dagger}$, using a translation procedure described in Appendix 4.8. An abstract-interpretation procedure is then applied to explore finite representations of the set of traces, using Kleene's 3-valued logic to conservatively interpret formulae. The abstract-interpretation procedure

```
  public class Worker implements Runnable {
    Request request;
    Resource resource; ...
    public void run() { ...
lw1   synchronized(resource) {
lwc     resource.processRequest(request);
lw2   }
    }
  }
```

Figure 4.1: Java fragment for worker thread in a web server with no explicit scheduling.

essentially computes a greatest fixed-point over the set of traces, starting with an abstract trace that represents all possible infinite traces from an initial state, and gradually increasing the set of abstract traces and reducing the set of represented concrete traces. Finally, the formula $(\varphi)^\dagger$ is evaluated on all of the abstract traces in the fixed point. If $(\varphi)^\dagger$ is satisfied in all of them, then the original ETL formula φ must be satisfied by all (infinite) traces of the program. However, it may be the case that for some programs that satisfy the ETL specification, our analysis only yields "maybe".

4.2.3 Running Example

Consider a web server in which a new thread is dynamically allocated to handle each received `http` request. Each thread handles a single request, then terminates and is subject to garbage collection. Assume that worker threads compete for some exclusively shared resource, such as exclusive access to a data file. Fig. 4.1 shows fragments of a Java program that implements such a naive web server.

A number of properties for the naive web-server implementation are shown in Table 4.1 as properties P1–P4. For now you may ignore the formulae in the third column; these will become clear as ETL syntax is introduced in Section 4.3.

Due to the unbounded arrival of requests to the web server, and the fact that a thread is dynamically created for each request, absence of starvation (P2) does not hold in the naive implementation. To guarantee absence of starvation, we introduce a scheduler thread into the web server. The web server now consists of a listener thread (as before) and a queue of worker threads managed by the scheduler thread. The listener thread receives an `http` request, creates a corresponding worker thread, and places the new thread on a scheduling queue. The scheduler thread picks up a worker thread from the queue and starts its execution (which is still a very naive implementation).

When using a web server with a scheduler, a number of additional properties of interest exist, labeled P5–P8 (for additional properties of interest see [112]). Fig. 4.2 shows fragments of a web-server program in which threads use an explicit FIFO scheduler.

The ability of our framework to model explicit scheduling queues provides a mechanism for address-

```
  public class Scheduler
        implements Runnable {
    protected Queue schedQ;
    protected Resource resource; ...
    public void run() {
ls1   while(true) { ...
ls2     synchronized(resource) {
ls3     while(resource.isAcquired())
ls4       resource.wait();
          // may block until
          // queue not empty
ls5       worker=schedQ.dequeue();
ls6       worker.start();
        }
      }
    }
  }
```

```
  public class Listener
        implements Runnable {
    protected Queue schedQ; ...
    public void run() {
la1   while(true) { ...
la2     req=rqStream.readObject();
la3     worker=new Thread(new Worker(req));
la4     schedQ.enqueue(worker);
      ...  }
    }
  }
  public class Worker
        implements Runnable {
    Request req;
    Resource resource; ...
    public void run() {
lw1   synchronized(resource) { ...
lwc     resource.processRequest(req);
        resource.notifyAll();
lw2   }
    }
  }
```

Figure 4.2: Java code fragment for a web server with an explicit scheduler.

ing issues of fairness in the presence of dynamic allocation of threads. (Further discussion of fairness is beyond the scope of this chapter).

4.3 Trace-Based Evolution Semantics

In this section, we define a trace-based semantic domain for programs that manipulate unbounded amounts of dynamically allocated storage. To allow specifying temporal properties of such programs, we employ first-order modal logic [42]. Various such logics have been defined, and in general they can be given a *constant-domain* semantics, in which the domain of all worlds is fixed, or a *varying-domain* semantics, in which the domains of worlds can vary and domains of different worlds can overlap. In the most general setting, in both types of semantics an object can exist in more than a single world, and an equality relation is predefined to express global equality between individuals.

To model the semantics of languages such as Java, and to hide the implementation details of dynamic memory allocation, we use a semantics with varying domains. However, the semantics is deliberately restricted because of our intended application to program analysis. By design, our *evolution semantics* has a notion of equality in the presence of dynamic allocation and deallocation, without the need to update a predefined global-equality relation. Evolution semantics is adapted from Lewis's counterpart semantics [66]. In both evolution and counterpoint semantics, an individual *cannot* exist in more than

Pr.	Description	Formula
P1	mutual exclusion over the shared resource	$\Box\, \forall t_1, t_2{:}\, thread.(t_1 \neq t_2) \rightarrow \neg(at[lw_c](t_1) \wedge at[lw_c](t_2))$
P2	absence of starvation for worker threads	$\Box\, \forall t{:}\, thread.at[lw_1](t) \rightarrow \Diamond\, at[lw_c](t)$
P3	a thread only created when a request is received	$\Box(\forall t{:}\, thread.\neg \odot t) \vee (\forall t{:}\, thread.\neg \odot t)\, \mathcal{U}\, (\exists v{:}\, request.\, \odot v)$
P4	each request is followed by thread creation	$\Box\, \exists v{:}\, request.\, \odot v \rightarrow \Diamond\, \exists t{:}\, thread.\, \odot t$
P5	mutual exclusion of listener and scheduler over scheduling queue	$\Box\, \forall t_1, t_2{:}\, thread.(t_1 \neq t_2) \rightarrow \neg(at[ls_3](t_1) \wedge at[la_3](t_2))$
P6	each created thread is eventually inserted into the scheduling queue	$\Box\, \forall t{:}\, thread.\, \odot t \rightarrow \Diamond\, \exists q{:}\, queue.rval[head.next^*](q, t)$
P7	each scheduled worker thread was removed from the scheduling queue	$\Box\, \forall t{:}\, thread.at[lw_1](t) \rightarrow \neg\exists q{:}\, queue.rval[head.next^*](q, t)$
P8	each worker thread waiting in the queue eventually leaves the queue	$\exists q{:}\, queue.\, \Box\, \forall t{:}\, thread.\, (rval[head.next^*](q, t)) \rightarrow \Diamond\, \neg(rval[head.next^*](q, t))$

Table 4.1: Web server ETL specification using predicates of Table 4.2.

a single world; each world has its own domain, and domains of different worlds are non-intersecting. Under this model, equality need only be defined within a single world's boundary; individuals of different worlds are unequal by definition.To relate individuals of different worlds, an evolution mapping is defined; however, unlike Lewis, we are interested in an evolution mapping that is reflexive, transitive, and symmetric, which models the fact that, during a computation, an allocated memory cell does not change its identity until deallocated. In Section 4.5.3, we show how to track statically, in the presence of abstraction, the equivalence relation induced by the evolution mapping.

As is often done, we add a skip action from the exit of the program to itself, so that all terminating traces are embedded in infinite traces. The semantics of the program is its set of infinite traces.

In the rest of this chapter, we work with a fixed set of predicates (or vocabulary) $\mathcal{P} = \{eq, p_1, \ldots, p_k\}$. We denote by $\mathcal{P}^k$ the set of predicates from $\mathcal{P}$ with arity k.

Definition 4.3.1 (World) *A **world** (program configuration) is represented via a first-order logical structure $W = \langle U_w, \iota_w \rangle$, where U_w is the domain (universe) of the structure, and ι_w is the interpretation function mapping predicates to their truth values; that is, for each $p \in \mathcal{P}^k$, $\iota_w(p)\colon U_w^k \rightarrow \{0, 1\}$, such that for all $u \in U_w$, $\iota_w(eq)(u, u) = 1$, and for all $u_1, u_2 \in U_w$ such that u_1 and u_2 are distinct individuals $\iota_w(eq)(u_1, u_2) = 0$.*

Definition 4.3.2 (Trace) *A **trace** is an infinite sequence of worlds* $\pi_1 \xrightarrow{D_{\pi_1}, e_{\pi_1}, A_{\pi_2}} \pi_2 \xrightarrow{D_{\pi_2}, e_{\pi_2}, A_{\pi_3}} \ldots$, *where: (i) each world represents a global state of the program,* π_1 *is an initial state, and for each* π_i*, its successor world* π_{i+1} *is derived by applying a single program action to* π_i*; (ii)* $D_{\pi_i} \subseteq U_{\pi_i}$ *is the set of individuals deallocated at* π_i*, and* $A_{\pi_{i+1}} \subseteq U_{\pi_{i+1}}$ *is the set of individuals newly allocated at* π_{i+1}*; (iii) each pair of consecutive worlds* π_i, π_{i+1} *is related by a stepwise **evolution function**, a bijective renaming function* $e_{\pi_i} : U_{\pi_i} \setminus D_{\pi_i} \to U_{\pi_{i+1}} \setminus A_{\pi_{i+1}}$.

Extracting Trace Properties

To extract trace properties, we need a language that can relate information from different worlds in a trace. We define the language of evolution logic (ETL), which is a first-order linear temporal logic with transitive closure, as follows:

Definition 4.3.3 (ETL Syntax) *An* **ETL formula** *is defined by*

$$\begin{aligned}\varphi ::= \; & 0 | 1 | p(v_1, \ldots, v_n) | \odot v_1 | \oslash v_1 | \varphi_1 \vee \varphi_2 | \neg\varphi_1 | \exists v_1 . \varphi_1 | (TC\ v_1, v_2 \colon \varphi_1)(v_3, v_4) \\ & | \varphi_1 \mathcal{U} \varphi_2 | \bigcirc \varphi_1\end{aligned}$$

where v_i *are logical variables.*

The set of free variables in a formula φ *denoted by* $FV(\varphi)$ *is defined as usual. In a transitive closure formula,* $FV((TC\ v_1, v_2 \colon \varphi_1)(v_3, v_4)) = (FV(\varphi_1) \setminus \{v_1, v_2\}) \cup \{v_3, v_4\}$.

The operators $\odot$ and $\oslash$ allow the specification to refer to the exact moments of birth and death (respectively) of an individual.[1]

Shorthand Formulae: For convenience, we also allow formulae to contain the shorthand notations $(v_1 = v_2) \triangleq eq(v_1, v_2)$, $(v_1 \neq v_2) \triangleq \neg eq(v_1, v_2)$, $\varphi_1 \wedge \varphi_2 \triangleq \neg(\neg\varphi_1 \vee \neg\varphi_2)$, $\varphi_1 \to \varphi_2 \triangleq \neg\varphi_1 \vee \varphi_2$, $\forall v . \varphi_1 \triangleq \neg(\exists v . \neg\varphi_1)$, $\Diamond \varphi_1 \triangleq 1 \mathcal{U} \varphi_1$, and $\Box \varphi_1 \triangleq \neg(1 \mathcal{U} \neg\varphi_1)$. We also use the shorthand $p^*(v_3, v_4)$ for $(TC\ v_1, v_2 \colon p(v_1, v_2))(v_3, v_4) \vee (v_3 = v_4)$, when p is a binary predicate.

In our examples, the predicates that record information about a single world include the predicates of Table 4.2, plus additional predicates defined in later sections. The set of predicates $\{at[lab](t) \colon lab \in Labels\}$ is parameterized by the set of program labels. Similarly, the set of predicates $\{rval[fld](o_1, o_2) \colon fld \in Fields\}$ is parameterized by the set of selector fields. We use the shorthand notation $rval[x.fld^*](v_1, v_2) \triangleq \exists v' . rval[x](v_1, v') \wedge rval[fld]^*(v', v_2)$. The transitive closure allows specifying properties relating to unbounded length of heap-allocated data structures (e.g., in $rval[fld]^*(v', v_2)$).

As in Chapter 2, we use unary predicates, such as $is\_thread(t)$, to represent type information. This could have been expressed using a many-sorted logic, but we decided to avoid this for expository purposes. Instead, for convenience we define the shorthands $\exists v \colon type . \varphi \triangleq \exists v . is\_type(v) \wedge \varphi$ and $\forall v \colon type . \varphi \triangleq \forall v . is\_type(v) \to \varphi$.

[1] These operators could be extended to handle allocation and deallocation of a (possibly unbounded) set of individuals.

Predicates	**Intended Meaning**
$is\_T(v)$	v is an object of type T
$\{at[lab](t) : lab \in Labels\}$	thread t is at label lab
$\{rval[fld](o_1, o_2) : fld \in Fields\}$	field fld of the object o_1 points to the object o_2
$heldBy(l, t)$	the lock l is held by the thread t
$blocked(t, l)$	the thread t is blocked on the lock l
$waiting(t, l)$	the thread t is waiting on the lock l

Table 4.2: Predicates used to record information about a single world.

Example 4.3.4 *Property P2 of Table 4.1 specifies the absence of starvation for worker threads (Fig. 4.1). The formula* $\exists t\colon thread.\ \Diamond\, at[lw_c](t)$ *states that some thread eventually enters the critical section. The formula* $\Box\, \exists t\colon thread.\ \Diamond\, at[lw_c](t)$ *expresses the fact that globally some thread eventually enters the critical section.*

The property $\Box(\forall v.\ \odot\, v \rightarrow \Diamond \oslash v)$ *states that globally, each individual that is allocated during program execution is eventually deallocated. Note that the universal quantifier quantifies over individuals of the world in which it is evaluated. This property is an instance of the commonly used "Response structure" [69, 37], in which an allocation in a world has a deallocation response in some future world.*

The properties

$$\forall t\colon thread.\ \Box(at[l_{lh}](t) \rightarrow \exists v.rval[i.next^*](t, v) \wedge \Diamond(at[l_{lh}](t) \wedge \neg rval[i.next^*](t, v)))$$
$$\forall t\colon thread.\ \Box(\forall v.at[l_{lh}](t) \wedge \neg rval[i.next^*](t, v) \rightarrow \Box\, \neg at[l_{lh}](t) \vee \neg rval[i.next^*](t, v))$$

establish a ranking function for linked data structures based on transitive reachability. These properties state that at the loop head l_{lh}*, the set of individuals transitively reachable from program variable* `i` *decreases on each iteration of the loop. (Typically* `i` *is a pointer that traverses a linked data structure during the loop.) Note that these properties relate an unbounded number of individuals of one world to another.*

The property $\Box(\forall v.\ \Diamond\, \Box\, \forall t\colon thread. \bigwedge_{\substack{x \in Var \\ fld \in Fields}} \neg rval[x.fld^*](t, v) \rightarrow \Diamond \oslash v)$ *is a desired property of a garbage collector — that all non-reachable items are eventually collected.*

Evolution Semantics

In the following definitions, $head(\pi)$ denotes the first world in a trace π, $tail(\pi)$ denotes the suffix of π without the first world, and π^i denotes the suffix of π starting at the i-th world. We also use $last(\tau)$ to denote the last world of a finite trace prefix τ.

Definition 4.3.5 (Evolution mapping) *Let τ be the finite prefix of length k of the trace π. We say that an individual $u \in U_{head(\tau)}$* ***evolves into*** *an individual $u' \in U_{last(\tau)}$ in the trace π in k steps, and write*

$\pi \models_k u \rightsquigarrow u'$ *when there is a sequence of individuals* $u_1, \ldots, u_k$ *such that* $u_1 = u$ *and* $u_k = u'$ *and for each two successive worlds in* τ, $u_{i+1} = e_{\tau_i}(u_i)$.

Definition 4.3.6 (Assignment evolution) *Let* τ *be the finite prefix of length* k *of the trace* π. *Given a formula* φ *and an assignment* Z *mapping free variables of* φ *to individuals of a domain* $U_{head(\tau)}$, *we say that* $\pi \models_k Z \rightsquigarrow Z'$ *(*Z ***evolves into*** Z' *in* π *in* k *steps) if for each free variable* fv_i *of* φ, $\pi \models_k Z(fv_i) \rightsquigarrow Z'(fv_i)$, $Z(fv_i) \in U_{head(\tau)}$, *and* $Z'(fv_i) \in U_{last(\tau)}$.

Definition 4.3.7 (ETL evolution semantics) *We define inductively when an ETL formula* φ *is satisfied over a trace* π *with an assignment* Z *(denoted by* $\pi, Z \models \varphi$*) as follows:*

- $\pi, Z \models \mathbf{1}$, *and not* $\pi, Z \models \mathbf{0}$.
- $\pi, Z \models p(v_1, \ldots, v_k)$ *when* $\iota_{head(\pi)}(p)(Z(v_1), \ldots, Z(v_k)) = 1$
- $\pi, Z \models \neg\varphi$ *when not* $\pi, Z \models \varphi$
- $\pi, Z \models \varphi \vee \psi$ *when* $\pi, Z \models \varphi$ *or* $\pi, Z \models \psi$
- $\pi, Z \models \exists v.\varphi(v)$ *when there exists* $u \in U_{head(\pi)}$ *s.t.* $\pi, Z[v \mapsto u] \models \varphi(v)$
- $\pi, Z \models (TC\ v_1, v_2 \colon \varphi)(v_3, v_4)$ *when there exists* $u_1, \ldots, u_{n+1} \in U_{head(\pi)}$, *such that* $Z(v_3) = u_1, Z(v_4) = u_{n+1}$, *and for all* $1 \leq i \leq n$,
  $\pi, Z[v_1 \mapsto u_i, v_2 \mapsto u_{i+1}] \models \varphi$.
- $\pi, Z \models \odot v$ *when* $Z(v) \in A_{head(tail(\pi))}$.
- $\pi, Z \models \oslash v$ *when* $Z(v) \in D_{head(\pi)}$.
- $\pi, Z \models \bigcirc\varphi$ *when there exists* Z' *such that* $tail(\pi), Z' \models \varphi$ *and* $\pi \models_1 Z \rightsquigarrow Z'$.
- $\pi, Z \models \varphi \mathcal{U} \psi$ *when there exists* $k \geq 1$, Z', *and* Z'' *s.t.,*
  $\pi^k, Z' \models \psi$ *and* $\pi \models_k Z \rightsquigarrow Z'$
  and for all $1 \leq j < k$, $\pi^j, Z'' \models \varphi$ *and* $\pi \models_j Z \rightsquigarrow Z''$,

We write $\pi \models \varphi$ *when* $\pi, Z \models \varphi$ *for every assignment* Z.

It is worth noting that the first-order quantifiers in this definition only range over the individuals of a single world, yet the overall effect achieved by using the evolution mapping is the ability to reason about individuals of different worlds, and how they relate to each other. In essence, the assignment $Z[v \mapsto u]$ binds v to (the evolution of) an individual from the domain of the world over which the quantifier was evaluated (cf. the semantics of $\bigcirc$ and $\mathcal{U}$).

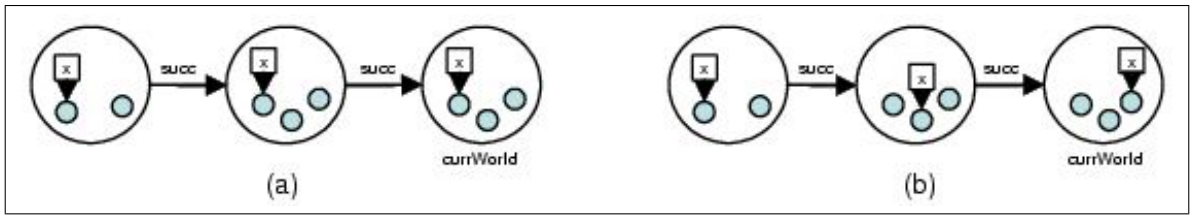


Figure 4.3: Interaction of first-order quantifiers and temporal operators.

The combination of first-order quantifiers and modal operators creates complications that do not occur in propositional temporal logics. In particular, the quantification domain of a quantifier may vary as the domain of the underlying worlds varies. Verification of ETL properties therefore requires a mechanism for recording the domain related to each quantifier, and for relating members of quantification domains to individuals of future worlds. For ETL, this mechanism is provided by evolution-mappings, which relate individuals of a world to the individuals of its successor world. Transitively composing evolution-mappings captures the evolution of individuals along a trace.

Example 4.3.8 *The formula* $\exists v.\ \Box\, x(v)$ *states that the pointer variable* `x` *remains constant throughout program execution, and points to an object that existed in the program's initial world. On the other hand, the formula* $\Box\, \exists v.x(v)$ *merely states that* `x` *never has the value* `null`*; however,* x *is allowed to point to different objects at different times in the program's execution, and in particular* x *can point to objects that did not exist in the initial world. Examples illustrating the two situations are shown in Fig. 4.3, where in* (a) x *points to the same object in all worlds, and in* (b) *it points to different objects in different worlds.*

Definition 4.3.9 *We say that a program* ***satisfies*** *an ETL formula* φ *when all (infinite) traces of the program satisfy* φ*.*

The evolution semantics allows each world to have a different domain, thus conceptually representing a varying-domain semantics, which allows dynamic allocation and deallocation of objects and threads. In Section 4.4, we give a possible implementation of this semantics in terms of evolving first-order logical structures.

Separable Specifications

It is interesting to consider subclasses of ETL for which the verification problem is somewhat easier. Two such classes are: (i) *spatially separable specifications* — do not place requirements on the relationships between individuals of one world; this allows each individual to be considered separately, and the verification problem can be handled as a set of propositional verification problems; (ii) *temporally separable specifications* — do not relate individuals across worlds. Essentially, this corresponds to the

Predicate	Intended Meaning
$world(w)$	w is a world
$currWorld(w)$	w is the current world
$initialWorld(w)$	w is the initial world
$succ(w_1, w_2)$	w_2 is the successor of w_1

Predicate	Intended Meaning
$exists(o, w)$	object o is in world w
$evolution(o_1, o_2)$	object o_1 evolves to o_2
$isNew(o)$	object o is new
$isFreed(o)$	object o is freed

Table 4.3: Trace predicates.

extraction of propositional information from each world, and having temporal specifications over the extracted propositions. This class was addressed in [20, 118].

4.4 Expressing Trace Semantics using First-Order Logic

In this section, we use first-order logic to express a trace semantics; we encode temporal operators using standard first-order quantifiers. This allows us to automatically derive an abstract semantics in Section 4.5. This approach also extends to other kinds of temporal logic, such as the μ-calculus. Our initial experience is that we are able to demonstrate that some temporal properties, including liveness properties, hold for programs with dynamically allocated storage.

4.4.1 Representing Infinite Traces via First-Order Structures

We encode a trace via an infinite first-order logical structure using the set of designated predicates specified in Table 4.3. Successive worlds are connected using the $succ$ predicate. Each world of the trace may contain an arbitrary number of individuals. The predicate $exists(o, w)$ relates an individual o to a world w in which it exists. Each individual only exists in a single world. The $evolution(o_1, o_2)$ predicate relates an individual o_1 to its counterpart o_2 in a successor world. The predicates $isNew$ and $isFreed$ hold for newly created or deallocated individuals and are used to model the allocation and deallocation operators.

Definition 4.4.1 (Concrete trace) *A **concrete trace** is a trace encoded as an infinite first-order logical structure $T = \langle U_T, \iota_T \rangle$, where U_T is the domain of the trace, and ι_T is the interpretation function mapping predicates to their truth value in the logical structure, i.e., for each $p \in \mathcal{P}^k$, $\iota_T(p) \colon U_T^k \to \{0, 1\}$. To exclude structures that cannot represent valid traces, we impose certain integrity constraints [91]. For example, we require that each world has at most one successor (predecessor), and that equality (eq) is reflexive.*

Example 4.4.2 *Fig. 4.4 shows four worlds of the trace $T_{4.4}^{\natural}$ where each world is depicted as a large node containing other nodes, and worlds along the trace are related by successor edges. Information*

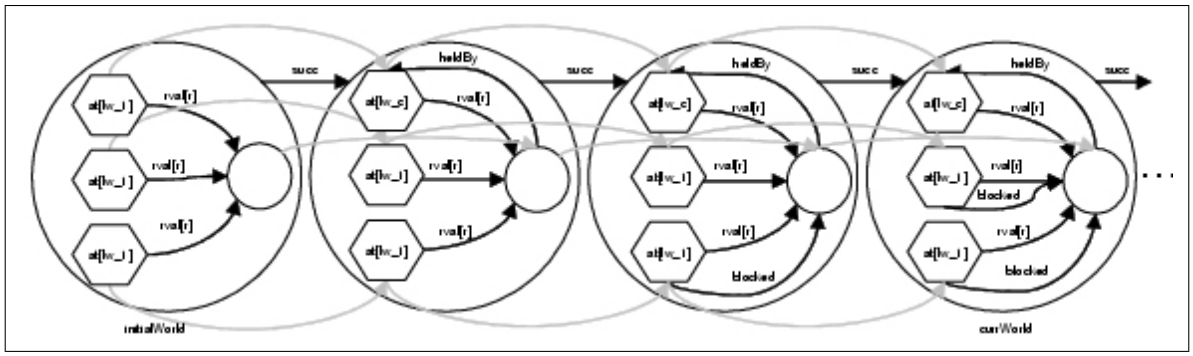


Figure 4.4: A concrete trace $T^{\natural}_{4.4}$.

in a single world is represented by a first-order logical structure, which is shown as a directed graph. Each node of the graph corresponds to a heap-allocated object. Hexagonal nodes correspond to thread objects, and small round nodes to other types of heap-allocated objects. Predicates holding for an object are shown inside the object node, and binary predicates are shown as edges. For brevity, we use the label $rval[r]$ to stand for $rval[resource]$. Gray edges, crossing world boundaries, are evolution edges, which relate objects of different worlds. Note that these are the only edges that cross world boundaries.

4.4.2 Exact Extraction of Trace Properties

Once traces are represented via first-order logical structures, trace properties can be extracted by evaluating formulae of first-order logic with transitive closure.

We translate a given ETL formula φ to an FO^{TC} formula $(\varphi)^{\dagger}$ by making the underlying trace structure explicit, and translating temporal operators to FO^{TC} claims over worlds of the trace. The translation procedure is straightforward, and given in Appendix 4.8.

Example 4.4.3 *The property $\exists t\colon thread.\ \Diamond\, at[lw_c](t)$ of Example 4.3.4 is translated to*

$$\exists w : world.\exists t : thread.initialWorld(w) \wedge exists(t,w) \wedge \exists w' \exists t' : thread.succ^*(w,w') \wedge exists(t',w') \wedge evolution^*(t,t') \wedge at[lw_c](t')$$

which evaluates to 1 *for the trace prefix of Fig. 4.4.*

Definition 4.4.4 *The **meaning** of a formula φ over a concrete trace T, with respect to an assignment Z, denoted by $[\![\varphi]\!]^T_2(Z)$, yields a truth value in $\{0,1\}$. The meaning of φ is defined inductively as follows:*

- $[\![l]\!]^T_2(Z) = l$ *(where $l \in \{0,1\}$)*
- $[\![p(v_1,\ldots,v_k)]\!]^T_2(Z) = \iota^T(p)(Z(v_1),\ldots,Z(v_k))$
- $[\![\varphi_1 \vee \varphi_2]\!]^T_2(Z) = \max([\![\varphi_1]\!]^T_2(Z), [\![\varphi_2]\!]^T_2(Z))$

- $[\![\neg\varphi_1]\!]_2^T(Z) = 1 - [\![\varphi_1]\!]_2^T(Z)$
- $[\![\exists v_1.\varphi_1]\!]_2^T(Z) = \max_{u \in U^T} [\![\varphi_1]\!]_2^T(Z[v_1 \mapsto u])$
- $[\![(TC\ v_1, v_2 \colon \varphi_1)(v_3, v_4)]\!]_2^T(Z) =$
$$max_{\substack{n \geq 1, u_1, \ldots, u_{n+1} \in U, \\ Z(v_3) = u_1, Z(v_4) = u_{n+1}}} \; min_{i=1}^{n} [\![\varphi_1]\!]_2^T(Z[v_1 \mapsto u_i, v_2 \mapsto u_{i+1}])$$

We say that T and Z ***satisfy*** *φ (denoted by $T, Z \models \varphi$) if $[\![\varphi]\!]_2^T(Z) = 1$. We write $T \models \varphi$ if for every Z we have $T, Z \models \varphi$.*

The correctness of the translation is established by the following theorem:

Theorem 4.4.5 *For every closed ETL formula φ and a trace π, $\pi \models \varphi$ if and only if $rep(\pi) \models (\varphi)^\dagger$, where $rep(\pi)$ is the first-order representation of π, i.e., the first-order structure that corresponds to π, in which every world in π is mapped to a world in $rep(\pi)$, with the $succ$ predicate holding for consecutive worlds.*

4.4.3 Semantics of Actions

Informally, a program action ac consists of a *precondition* ac_{pre}, expressed as a logical formula, under which the action is *enabled*, and a set of formulae for updating the values of predicates according to the effect of the action. An enabled action specifies that a possible next world in the trace is one in which the interpretations of every predicate p of arity k is determined by evaluating a formula $\varphi_p(v_1, v_2, \ldots, v_k)$, which may use $v_1, v_2, \ldots, v_k$ and all predicates in $\mathcal{P}$ (see [91]).

4.5 Exploring Finite Abstract Traces via Abstract Interpretation

In this section, we give an algorithm for conservatively determining the validity of a program with respect to an ETL property. A key difficulty in proving liveness properties is the fact that a liveness property might be violated only by an infinite trace. Therefore, our procedure for verifying liveness properties is a greatest fixed-point computation, which works down from an initial approximation that represents all infinite traces. In this section, we present our abstract-interpretation algorithm; procedure `explore` of Fig. 4.8.

Our approach uses finite representations of infinite traces. Finite representations are obtained by abstraction to three-valued logical structures. The third logical value, 1/2, represents "unknown" and may result from abstraction. The abstract semantics conservatively models the effect of actions on abstract representations.

4.5.1 A Finite Representation of Infinite Traces

The first step in making the algorithm of Fig. 4.8 feasible is to define a finite representation of sets of infinite traces. Technically, we use 3-valued logical structures to finitely represent sets of infinite traces. The construction in this section follows similar lines to the construction in Section 2.4, where instead of abstracting single configurations as in Section 2.4, in this section we abstract infinite traces.

Definition 4.5.1 (Abstract trace) *An **abstract trace** is a 3-valued first-order logical structure $T = \langle U_T, \iota_T \rangle$, where U_T is the domain of the abstract trace, and ι_T is the interpretation, mapping predicates to their truth values, i.e., for each $p \in \mathcal{P}^k$, $\iota_T(p) \colon U_T^k \to \{0, 1, 1/2\}$. We refer to the values 0 and 1 as **definite values**, and to $1/2$ as a **non-definite value**.*

*An individual u for which $\iota_T(eq)(u, u) = 1/2$ is called a **summary individual**;[2] a summary individual may represent more than one concrete individual.*

*The **meaning** of a formula φ over a 3-valued abstract trace T, with respect to an assignment Z, denoted by $[\![\varphi]\!]_3^T(Z)$, is defined exactly as in Def. 4.4.4, but interpreted over $\{0, 1, 1/2\}$.*

*We say that a trace T with an assignment Z **potentially satisfies** a formula φ when $[\![\varphi]\!]_3^T(Z) \in \{1, 1/2\}$ and denote this by $T, Z \models_3 \varphi$.*

We now define how concrete traces are represented by abstract traces (extending the concepts of Section 2.4 to work for traces). The idea is that each individual of a concrete trace is mapped by the abstraction into an individual of an abstract trace. The new two definitions permit an (abstract or concrete) trace to be related to a less-precise abstract trace. Abstraction is a special case of this in which the first trace is a concrete trace. First, the following definition imposes an order on truth values of the 3-valued logic:

Definition 4.5.2 *For $l_1, l_2 \in \{0, 1, 1/2\}$, we define the **information order** on truth values as follows: $l_1 \sqsubseteq l_2$ if $l_1 = l_2$ or $l_2 = 1/2$.*

The embedding ordering of abstract traces is then defined as follows:

Definition 4.5.3 (Trace embedding) *Let $T = \langle U, \iota \rangle$ and $T' = \langle U', \iota' \rangle$ be abstract traces encoded as first-order structures. A function $f \colon T \to T'$ such that f is surjective is said to **embed** T **into** T' if for each predicate $p \in \mathcal{P}^k$, and for each $u_1, \ldots, u_k \in U$:*

$$\iota(p(u_1, u_2, \ldots, u_k)) \sqsubseteq \iota'(p(f(u_1), f(u_2), \ldots, f(u_k)))$$

*We say that T' **represents** T when there exists such an embedding f.*

[2] Note that for all $u \in U_T$, $\iota_T(eq)(u, u) = 1$ or $\iota_T(eq)(u, u) = 1/2$.

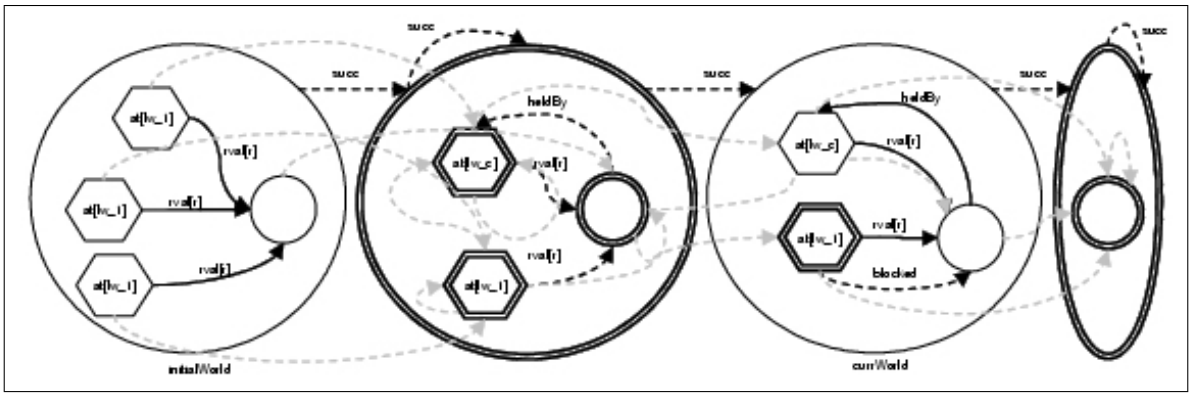


Figure 4.5: An abstract trace $T_{4.4}$ that represents the concrete trace $T^{\flat}_{4.4}$.

As in Section 2.4, we use *canonical abstraction* as an embedding function, but this time for traces. Recall that canonical abstraction maps individuals to an abstract individual based on the values of the individuals' unary predicates. All individuals having the same values for unary predicate symbols are mapped by the abstraction to the same abstract individual. We denote the canonical abstraction of a trace T by *blur*(T). Canonical abstraction guarantees that each abstract trace is no larger than some fixed size, known *a priori*.

Example 4.5.4 *Fig. 4.5 shows an abstract trace, with four abstract worlds, that represents the concrete trace of Fig. 4.4. An individual with double-line boundaries is a summary individual representing possibly more than a single concrete individual. Similarly, the worlds with double-line boundaries are summary worlds that possibly represent more than a single world. Dashed edges are* $1/2$ *edges, that represent relations that may or may not hold. For example, a* $1/2$ *successor edge between two worlds represents the possible succession of worlds. The summary world following the initial world represents the two concrete worlds between the initial and the current world of* $T^{\flat}_{4.4}$*, which have the same values for their unary predicates. Similarly, the summary node labeled* $at[lw_1]$ *represents all thread individuals in these worlds that reside at label* lw_1.

Note that this abstract trace also represents other concrete traces besides $T^{\flat}_{4.4}$*, for example, concrete traces in which in the current world some threads are blocked on the lock and some are not blocked.*

4.5.2 Abstract Interpretation

The abstract semantics represents abstract traces using 3-valued structures. Intuitively, applying an action to an abstract trace unravels the set of possible next successor worlds in the trace. That is, an abstract action elaborates an abstract trace by materializing a world w from the summary world at the tail of the trace; w becomes the definite successor of the current world $currWorld$, and w's (indefinite) successor is the summary world at the tail of the trace. $currWorld$ is then advanced to w, which often

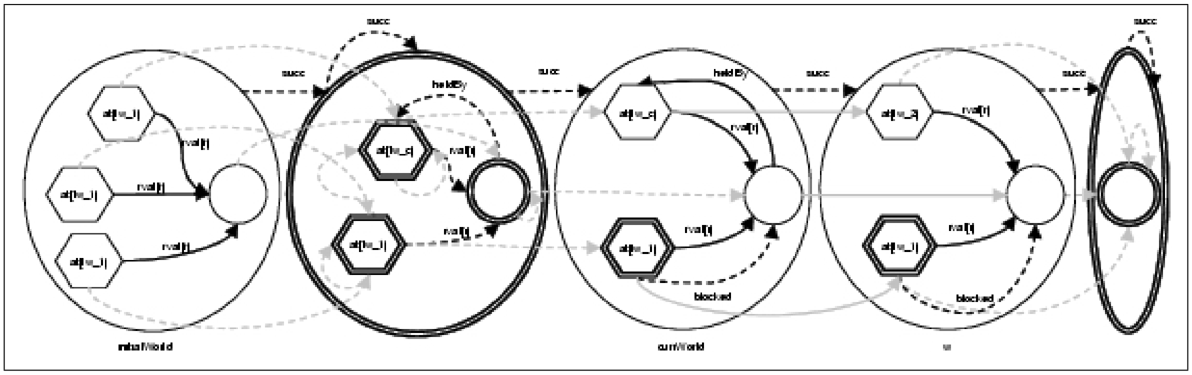


Figure 4.6: An intermediate abstract trace, which represents the first stage of applying an action to $T_{4.4}$.

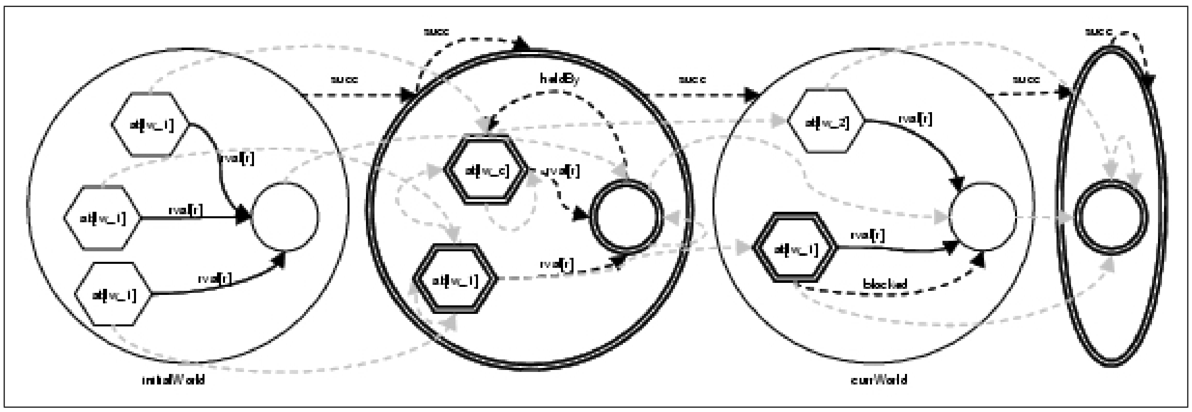


Figure 4.7: The resulting abstract trace after applying an action over $T_{4.4}$ (after advancing $currWorld$).

causes the former $currWorld$ to be merged with its predecessor. When a trace is extended, we evaluate the formula's precondition and its update formulae using 3-valued logic (as in Def. 4.5.1).

Example 4.5.5 *Figures 4.5, 4.6, and 4.7 illustrate the application of the action that releases a lock. Fig. 4.6 shows the materialization of the next successor world for the trace $T_{4.4}$ of Fig. 4.5. In the successor world, the thread that was at label lw_c no longer holds the lock and has advanced to label lw_2. The $currWorld$ predicate is then advanced, and the former $currWorld$ is merged with its predecessor, resulting in the abstract trace shown in Fig. 4.7.*

The abstract-interpretation procedure `explore` is shown in Fig. 4.8. It computes a greatest fixed-point starting with the set $\{T_1^\top, T_2^\top\}$; these two abstract traces represent all possible concrete (infinite) traces that start at a given initial state. $T_1^\top$ and $T_2^\top$ each have two worlds: an initial world that represents the initial program configuration connected by a 1/2-valued successor edge to a summary world that represents the unknown possible suffixes. The summary world w_{s1} of $T_1^\top$ has a summary individual u_{s1} related to it. The summary individual u_{s1} has $1/2$ values for all of its predicates, including $exists(u_{s1}, w_{s1}) = 1/2$, meaning that future worlds of the trace do not necessarily contain any individ-

```
explore() {
 Traces = {T_1^⊤, T_2^⊤}
 while changes occur {
  select and remove τ from Traces
  for each action ac enabled for τ
   Traces = Traces ⋃ {ac(τ)}
 }
 for each τ ∈ Traces
  if τ ⊭_3 (φ)^† report possible error
}
```

Figure 4.8: Computing the set of abstract traces and evaluating the property $(\varphi)^\dagger$.

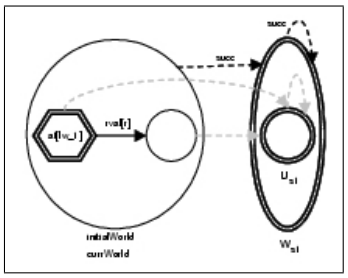


Figure 4.9: An initial abstract trace $T_1^\top$.

uals. The summary world of $T_2^\top$ has no summary individual related to it and represents suffixes in which all future worlds are empty. Fig. 4.9 shows an initial abstract trace (corresponding to $T_1^\top$) representing all traces starting with an arbitrary number of worker threads at label lw_1 sharing a single lock.

The procedure `explore` accumulates abstract traces in the set $Traces$ until a fixed point is reached. Throughout this process, however, the set of concrete traces represented by the abstract traces in $Traces$ is actually decreasing. It is in this sense that `explore` is computing a greatest fixed-point.

Once a fixed point has been reached, the property of interest is evaluated over the abstract traces in the fixed point. Formula evaluation over an abstract trace exploits values of instrumentation predicates when possible (this is explained in the following section). This allows the use of recorded definite values, whereas re-evaluation might have yielded $1/2$.

We now show the soundness of the approach. We extend mappings on individuals to operate on assignments: If $f\colon U^T \to U^{T'}$ is a function and $Z\colon Var \to U^T$ is an assignment, $f \circ Z$ denotes the assignment $f \circ Z\colon Var \to U^{T'}$ such that $(f \circ Z)(v) = f(Z(v))$. One of the nice features of 3-valued logic is that the soundness of the analysis is established by the following theorem (which generalizes [91] for the infinite case):

Theorem 4.5.6 *[Embedding Theorem] Let $T = \langle U^T, \iota^T\rangle$ and $T' = \langle U^{T'}, \iota^{T'}\rangle$ be two traces encoded as first-order structures, and let $f\colon U^T \to U^{T'}$ be a function such that $T \sqsubseteq^f T'$. Then, for every formula φ and complete assignment Z for φ, $[\![\varphi]\!]_3^T(Z) \sqsubseteq [\![\varphi]\!]_3^{T'}(f \circ Z)$.*

The algorithm in Fig. 4.8 must terminate. Furthermore, whenever it does not report an error, the program satisfies the original ETL formula φ.

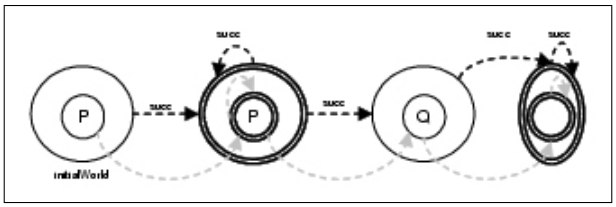


Figure 4.10: $\exists v.P(v)\mathcal{U}Q(v)$ holds in all concrete traces that the abstract trace $T_{4.10}$ represents, yet $\exists v.P(v)\mathcal{U}Q(v)$ evaluates to $1/2$ on $T_{4.10}$ itself.

Predicate	Intended Meaning	Formula
$twe(o_1, o_2)$	object o_1 is equal to object o_2 possibly across worlds	$(o_1 = o_2) \vee evolution^*(o_1, o_2)$ $\vee evolution^*(o_2, o_1)$
$current(o)$	object o is a member of current world	$\exists w\colon world(o, w) \wedge currWorld(w)$

Table 4.4: Trace instrumentation predicates.

It often happens that this approach to verifying temporal properties yields $1/2$, due to an overly conservative approximation. In the next section, we present machinery for refining the abstraction to allow successful verification in interesting cases.

Example 4.5.7 *For clarity and ease of presentation, we use an artificial example, which is also used in the next section. Fig. 4.10 shows an abstract trace in which the property* $\exists v.P(v)\mathcal{U}Q(v)$ *holds for all the concrete traces represented by the abstract trace, but the formula* $\exists v.P(v)\mathcal{U}Q(v)$ *evaluates to* $1/2$ *because the successor and evolution edges have value* $1/2$.

4.5.3 Property-Guided Instrumentation

To refine the abstraction, we can maintain more precise information about the correctness of temporal formulae as traces are being constructed. This principle is referred to in [91] as the *Instrumentation Principle*. This work goes beyond what was mentioned there, by showing how one could actually obtain instrumentation predicates from the temporal specification.

Trace Instrumentation

The predicates in Table 4.4 are required for preserving properties of interest under abstraction. The instrumentation predicate $current(o)$ denotes that o is a member of the current world and should be distinguished from individuals of predecessor worlds. This predicate is required due to limitations of canonical embedding. The predicate $twe(o_1, o_2)$ records equality across worlds and is required due to the loss of information about concrete locations caused by abstraction.

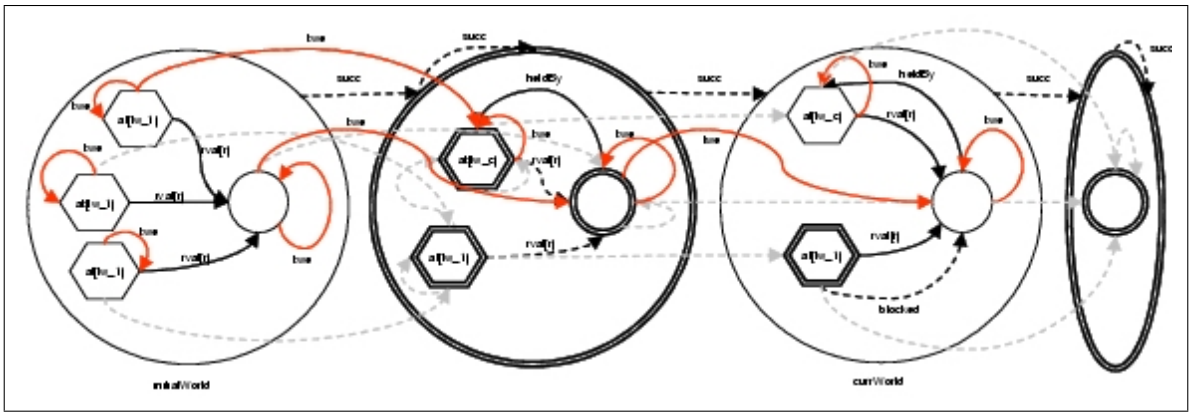


Figure 4.11: Abstract trace with transworld equality instrumentation (Only 1-valued transworld equality edges are shown).

Transworld Equality: In the evolution semantics, two individuals are considered to be different incarnations of the same individual when one may transitively evolve into the other. We refer to this notion of equality as *transworld equality* and introduce an instrumentation predicate $twe(v_1, v_2)$ to capture this notion.

Because the abstraction operates on traces (and not only single worlds), individuals of different worlds may be abstracted together. Transworld equality is crucial for distinguishing a summary node that represents different incarnations of the same individual in different worlds from a summary node that may represent a number of different individuals.

Transworld equality is illustrated in Fig. 4.11; the 1-valued twe self-loop to the summary thread-node at label lw_c records the fact that this summary node actually represents multiple incarnations of a single thread, and not a number of different threads.

Temporal Instrumentation

Given an ETL specification formula, we construct a corresponding set of instrumentation predicates for refining the abstraction of the trace according to the property of interest. The set of instrumentation predicates corresponds to the sub-formulae of the original specification.

Example 4.5.8 *In Example 4.5.7, the property $\exists v.P(v)\,\mathcal{U}\,Q(v)$ evaluated to $1/2$ although it is satisfied by all concrete traces that $T_{4.10}$ represents. We now add the temporal instrumentation predicates $I_p(v)$ and $I_q(v)$ to record the values of the temporal subformulae $P(v)$ and $Q(v)$. The predicates are updated according to their value in the previous worlds. Note the use of transworld equality instrumentation to more precisely record transitive evolution of objects. In particular, this provides the information that the summary node of the second world is an abstraction of different incarnations of the same single object. This is shown in Fig. 4.12.*

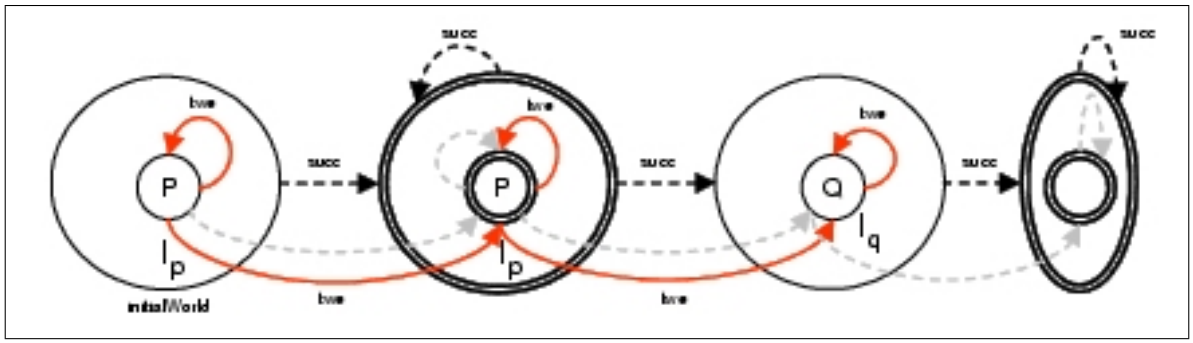


Figure 4.12: In the abstract trace $T_{4.12}$, $\exists v.P(v)\mathcal{U}Q(v)$ evaluates to 1.

4.6 Related Work

The Bandera Specification Language (BSL) [20] allows writing specifications via common high-level patterns. In BSL, it is impossible to relate individuals of different worlds, and impossible to refer to the exact moments of allocation and deallocation of an object.

In [81], a special case of the abstraction from [115, 118], named "counter abstraction", is used to abstract an infinite-state parametric system into a finite-state one. They use static abstraction, i.e., they have a preceding model-extraction phase. In contrast, in our work abstraction is applied dynamically on every step of state-space exploration, which enables us to handle dynamic allocation and deallocation of objects and threads.

In [118], we have used observing-propositions defined over a first-order configuration to extract a propositional Kripke structure from a first-order one. The extracted structure was then subject to PLTL model-checking techniques. This approach is rather limited, because individuals of different worlds cannot be specifically related.

4.7 Conclusion

We believe this work provides a foundation for specifying and verifying properties of programs manipulating the heap with dynamic allocation and deallocation of objects and threads. In the next chapter, we develop a more scalable approach for verification of ETL properties.

4.8 Translation of ETL to FO^{TC}

We say that an ETL sub-formula is temporally-bound if it appears under a temporal operator. Translations for temporally-bound and non-temporally-bound formulae are different, since non-temporally-bound formulae should be bound to the initial world of the trace.

Definition 4.8.1 *[ETL translation to FO^{TC}] We denote by $(\varphi)^{\dagger w}$ the bounded translation of a formula φ in a world w and by $(\varphi)^{\dagger}$ the non-bounded translation.*

- $(\varphi)^{\dagger} = \exists w\colon world.initialWorld(w) \wedge (\varphi)^{\dagger w}$
- *if φ is an atomic formula other than $\odot x$ and $\oslash x$ then $(\varphi)^{\dagger w} = \varphi$. If $\varphi = \odot x$ then $(\varphi)^{\dagger w} = isNew(x)$. If $\varphi = \oslash x$ then $(\varphi)^{\dagger w} = isFreed(x)$.*
- $(\varphi \wedge \psi)^{\dagger w} = (\varphi)^{\dagger w} \wedge (\psi)^{\dagger w}$, $(\varphi \vee \psi)^{\dagger w} = (\varphi)^{\dagger w} \vee (\psi)^{\dagger w}$, $(\neg\varphi)^{\dagger w} = \neg(\varphi)^{\dagger w}$
- $(\exists x\ \varphi)^{\dagger w} = \exists x.exists(w, x) \wedge (\varphi)^{\dagger w}$
- $((TC\ x_1, x_2\colon \varphi)(x_3, x_4))^{\dagger w} = (TC\ x_1, x_2\colon (\varphi)^{\dagger w} \wedge exists(w, x_1) \wedge exists(w, x_2))(x_3, x_4)$
- $(\varphi(x_1, \ldots, x_n)\mathcal{U}\psi(y_1, \ldots, y_k))^{\dagger w} =$
$$\exists w'\colon world.\exists y'_1, \ldots, y'_k.succ^*(w, w') \wedge (\psi(y'_1, \ldots, y'_k))^{\dagger w'}$$
$$\wedge \textstyle\bigwedge_{1 \le i \le k} evolution^*(y_i, y'_i) \wedge \forall \tilde{w}\colon world.\exists x'_1, \ldots, x'_n.(succ^*(w, \tilde{w})$$
$$\wedge succ^*(\tilde{w}, w') \rightarrow (\varphi(x'_1, \ldots, x'_n))^{\dagger \tilde{w}} \wedge \textstyle\bigwedge_{1 \le j \le n} evolution^*(x_j, x'_j))$$
- $(\bigcirc\varphi(x_1, \ldots, x_n))^{\dagger w} =$
$$\exists w'\colon world.\exists x'_1, \ldots, x'_n.succ(w, w')$$
$$\wedge (\varphi(x'_1, \ldots, x'_n)^{\dagger w'} \wedge \textstyle\bigwedge_{1 \le j \le n} evolution(x_j, x'_j) \wedge exists(x'_j, w')$$

Note that x_i and y_i are not necessarily distinct. Simplified translations may be used for the $\Diamond$ and $\Box$ temporal operators.

Chapter 5

Automatic Verification of Temporal Heap Properties

In this chapter, we present a framework for verifying temporal properties of sequential and concurrent heap-manipulating programs. The specification language we use is evolution temporal logic (ETL), a first-order linear temporal logic, that allows us to naturally specify safety and liveness properties of programs that dynamically allocate objects and threads without an a priori bound. Our framework is able to verify both safety and liveness properties of such programs.

In this chapter, we define a state-based constant-domain semantics for ETL. This yields a verification algorithm that is more efficient than the one described in Chapter 4.

The best thing about the future is that it comes only one day at a time.
–Abraham Lincoln.

5.1 Introduction

Two of the main challenges of software verification are handling heap-allocated storage and handling dynamic allocation of objects and threads. These features are often ignored or handled in an imprecise manner by existing verification and static-analysis approaches, especially for concurrent programs [87].

In this chapter, we present a framework for verifying temporal properties of sequential and concurrent heap-manipulating programs. Unlike many of the existing verification techniques, our framework does not impose an a priori bound on the number of allocated objects and threads. The specification language we use is a first-order linear temporal logic that allows us to naturally specify safety and liveness properties of programs that dynamically allocate objects and threads. This should be contrasted with traditional model-checking techniques that use propositional temporal logic [17].

Verifying that a property φ holds for a program requires verifying that φ holds on all program traces.

```
   public Set mark(Element root) {
m1    Set pending = new HashSet();
m2    Set marked = new HashSet();
m3    if (root != null) {
m4       pending.add(root);
m5       while (!pending.isEmpty()) {
m6          Element x = (Element) pending.iterator().next();
m7          pending.remove(x);
m8          marked.add(x);
m9          Element t = x.left;
m10         if (t != null && !marked.contains(t))
m11               pending.add(t);
m12         t = x.right;
m13         if (t != null && !marked.contains(t))
m14               pending.add(t);
m15      }
m16   }
m17   return marked;
   }
```

Figure 5.1: Java source for the mark-phase procedure.

In this chapter, we use abstract interpretation to conservatively check if there exists a *violation trace* — a trace that satisfies the *violation property* $\neg\varphi$.

Our verification method is *sound*, that is, if we say that a violation trace does not exist, the property is guaranteed to hold on all program traces. However, since we over-approximate the set of program traces, our framework may yield false alarms, i.e., it may say that there exists a trace satisfying the violation property when there is no such execution trace of the program.

Motivating Example

As a motivating example, consider the marking phase procedure of Fig. 5.1. This procedure is part of a "stop-the-world" (non-incremental) mark-and-sweep garbage collector [110] in which garbage collection is performed as a single atomic step while the program is halted[1]. For simplicity, we assume that the heap has a single root, that all objects have `right` and `left` fields, and that there are no arrays of references. We would like to verify the following safety and liveness properties for this procedure:

(P1) all nodes reachable from the root are eventually marked

(P2) all marked nodes are reachable from the root

[1] An optimized version of this GC algorithm is implemented in Sun's JVM1.2.

Verifying these properties is a challenging task, as there is no bound on the number of objects in the heap, and they have no static names. In the following sections, we will see how to formulate these properties in our framework and how they are verified.

Related Work

Typestate Checking

A *typestate property* is a safety property that defines the valid sequences of operations that could be performed on each object of the specified type. One of the open challenges in typestate verification is an adequate treatment of aliasing. Some approaches, such as the original work on typestate [103], forbid any aliasing. Other approaches (e.g., [72] and [27]) take a two-phased approach in which typestate analysis is preceded by an aliasing-analysis, which may result in a loss of precision.

Our work generalizes and extends previous work based on typestate checking in at least three significant aspects: (i) we handle properties involving more than a single object; (ii) we handle liveness properties; (iii) we handle aliasing with high precision and as part of the verification procedure rather than as a preceding phase.

Moreover, our framework can also be used to perform typestate checking for concurrent programs with dynamic allocation and deallocation of objects and threads.

Note that in contrast to Chapter 3, our goal in this chapter is to handle arbitrary programs (not necessarily shallow) and arbitrary ETL properties (generalizing typestate), but we do not guarantee precise results, and use techniques that are more expensive than the ones used in Chapter 3.

Model Checking Object-Based Software

Distefano et. al. [34] define a temporal logic for reasoning about object-based software. The abstraction used there is a limited one and their representation does not handle structural relationships among heap objects.

Previous Work

This chapter generalizes contributions presented in Chapter 2, Chapter 4, and Section 7.1 of this thesis. It generalizes our work on verifying strong safety properties (Chapter 2), verifying non-nested liveness properties (Chapter 4), and verifying local temporal safety properties (Section 7.1). All the properties discussed in these chapters can be formulated and verified in this framework. While Chapter 4 uses an expensive abstract interpretation of the trace-semantics, the framework presented in the current chapter works with a state-based semantics which is significantly more efficient. In terms of precision the two methods are incomparable: (i) the trace-based semantics records history even when it is not part of the temporal specification; (ii) the state-based semantics only tracks history relevant to the specification.

However, we generally expect the state-based semantics to behave better under abstraction. As a trivial example, the trace-based semantics will fail even on safety properties. Another key difference is that Chapter 4 uses varying-domain semantics while this chapter uses a constant-domain semantics (see Section 5.2.2).

Outline

The rest of this chapter is organized as follows. In Section 5.2, we define the syntax of ETL, and provide concrete trace-based and state-based constant-domain semantics. Section 5.3 then describes an abstract state-based semantics for ETL. In Section 5.4, we show how to encode the state-based semantics using first order logic providing a verification algorithm for ETL. Finally, we conclude with Section 5.5.

5.2 Evolution Temporal Logic

In this section, we define the syntax and semantics of evolution temporal logic (ETL) – a first-order linear temporal logic [69] that allows us to naturally specify properties of heap-manipulating programs. The semantics we define for ETL is a *constant-domain semantics* in which the universe of all configurations is fixed. In Section 5.2.2, we define a trace-based semantics for ETL. Section 5.2.3 then defines a state-based semantics whose abstract interpretation provides the abstract semantics of Section 5.3.2.

5.2.1 Syntax

The following defines ETL syntax in a way similar to Definition 4.3.3 with the following key differences: (i) we use a designated predicate ϵ to explicitly denote the existence of an individual in a configuration; (ii) we add temporal past operators.

Definition 5.2.1 (ETL Syntax) *An **ETL formula** over a **vocabulary** $\mathcal{P} = \{eq, \epsilon, p_1, \ldots, p_n\}$ is defined by*

$$\begin{aligned}\varphi ::= \;& 0 | 1 | p(v_1, \ldots, v_n) | \varphi_1 \vee \varphi_2 | \varphi_1 \wedge \varphi_2 | \neg\varphi_1 | \exists v_1.\varphi_1 | \forall v_1.\varphi_1 | (TC\ v_1, v_2 \colon \varphi_1)(v_3, v_4) \\ & | \varphi_1 \mathcal{U} \varphi_2 | \varphi_1 \mathcal{W} \varphi_2 | \bigcirc \varphi_1 | \ominus \varphi_1 | \varphi_1 \mathcal{S} \varphi_2 | \varphi_1 \mathcal{B} \varphi_2\end{aligned}$$

*where v_i are logical variables. The **vocabulary** $\mathcal{P}$ contains the special predicate symbol $eq(v_1, v_2)$ that denotes equality between individuals and the predicate symbol $\epsilon(v)$ that denotes the existence of an individual in the current state. We denote by $\mathcal{P}^k$ the predicates of arity k in $\mathcal{P}$.*

*The set of **free variables** in a formula φ, denoted by $FV(\varphi)$, is defined as usual. In a transitive closure formula $FV((TC\ v_1, v_2 \colon \varphi)(v_3, v_4)) = (FV(\varphi) \setminus \{v_1, v_2\}) \cup \{v_3, v_4\}$.*

*We refer to formulae of one of the forms $0, 1, p(v_1, \ldots, v_n)$ as **atomic formulae**. We refer to formulae of one of the forms $\varphi_1 \mathcal{U} \varphi_2$, $\varphi_1 \mathcal{W} \varphi_2$, $\bigcirc\varphi_1$, $\ominus \varphi_1$, $\varphi_1 \mathcal{S} \varphi_2$, and $\varphi_1 \mathcal{B} \varphi_2$ as **principally temporal***

Predicates	**Intended Meaning**
$\epsilon(v)$	v exists in the configuration
$eq(v_1, v_2)$	v_1 equals to v_2
$\{at[lab]() : lab \in Labels\}$	program is at label lab
$\{x(v) : x \in PVar\}$	object v is pointed-to by reference variable x
$\{fld(v_1, v_2) : fld \in RFields\}$	field fld of the object v_1 points to the object v_2
$marked(v)$	object v has been marked
$pending(v)$	object v is pending to be marked

Table 5.1: Predicates used for the example program.

***formulae**, and to any formula that does not contain a principally temporal formula as a **non-temporal formula**. We restrict the formulae in a transitive-closure operator to non-temporal formulae.*

We use the shorthand notations $\Diamond\varphi \triangleq (1\mathcal{U}\varphi)$, $\Box\varphi \triangleq \neg(1\mathcal{U}\neg\varphi)$. We also use the shorthand notations $\odot(v) \triangleq \epsilon(v) \wedge \ominus\neg\epsilon(v)$, and $\oslash(v) \triangleq \ominus\epsilon(v) \wedge \neg\epsilon(v)$, which refer to the exact moment in which an object is allocated and deallocated, respectively.

To ease the use of the ϵ predicate, which provides relativization of quantification and predicate values to individuals that actually exist in a configuration, we introduce the shorthand notations $p^{\epsilon}(v_1, \ldots, v_k) \triangleq \bigwedge_{1\leq i\leq k}\epsilon(v_i) \wedge p(v_1, \ldots, v_k)$ and $\neg p^{\epsilon}(v_1, \ldots, v_k) \triangleq \bigwedge_{1\leq i\leq k}\epsilon(v_i) \wedge \neg p(v_1, \ldots, v_k)$. In this section, we only use predicates over objects that actually exist and therefore assume all predicates (and negated predicates) are ϵ-relativized and omit the ϵ superscript from predicate symbols. Allowing non-relativized predicates (e.g., for handling allocation) is quite straightforward.

Finally, we use shorthand notations for quantifiers that only quantify over objects that actually exist in a configuration: $\exists^{\epsilon}v.\varphi(v) \triangleq \exists v.\epsilon(v) \wedge \varphi(v)$ and $\forall^{\epsilon}v.\varphi(v) \triangleq \forall v.\epsilon(v) \rightarrow \varphi(v)$.

The operator $\bigcirc$ is the *next* operator, and $\bigcirc\varphi$ requires the formula φ to hold in the next state of the trace. The operator $\ominus$ is the *previous* operator, and $\ominus\varphi$ requires φ to hold in the previous state of the trace. The operator $\mathcal{U}$ is the *until* operator, and $\varphi\mathcal{U}\psi$ requires that:(i) ψ eventually holds, and (ii) until that point φ holds. The operator $\mathcal{W}$ is the *weak until* operator, and $\varphi\mathcal{W}\psi$ requires that φ holds at least until ψ holds. Note that $\Box\varphi = \varphi\mathcal{W}0$. The operator $\mathcal{S}$ is the *since* operator, and $\varphi_1\mathcal{S}\varphi_2$ requires that φ_2 held some time in the past and since that point φ_1 holds. The operator $\mathcal{B}$ is the weak version of $\mathcal{S}$ and $\varphi_1\mathcal{B}\varphi_2$ requires that φ_1 holds since φ_2 held. A temporal formula that does not use $\bigcirc$,$\mathcal{U}$ or $\mathcal{W}$ is called a past formula. To allow incremental tableau construction, we restrict the use of past formulae to only appear within a single containing $\Box$ or $\Diamond$ operator (i.e., with no nesting of future operators).

The predicates used in this chapter are listed in Table 5.1. Excluding the predicates $marked(v)$ and $pending(v)$, all predicates of Table 5.1 are variations on the predicates used in Chapter 2, adapted for

sequential programs. We use unary predicates to represent values of reference variables and boolean properties of individuals. A unary predicate $x(v)$ holds for an individual v when it is referenced by the reference variable `x`. We use binary predicates to represent values of reference fields. A binary predicate $fld(v_1, v_2)$ holds when the field `fld` of v_1 points to v_2.

Example 5.2.2 *The properties of interest for the marking procedure are formulated as the following ETL formulae.*

$$\textit{(P1)}\ \ \forall^{\epsilon} v.\varphi_{r[root]}(v) \rightarrow \Diamond\, marked(v)$$
$$\textit{(P2)}\ \ \Box\, \forall^{\epsilon} v.marked(v) \rightarrow \varphi_{r[root]}(v)$$

where $\varphi_{r[root]}(v) = root(v) \vee \exists^{\epsilon} v_1.root(v_1) \wedge (TC\ w_1, w_2 \colon right(w_1, w_2) \vee left(w_1, w_2)))(v_1, v)$

In the running example, the unary predicate $root(v)$ holds for the individual that is referenced by the variable `root`. The unary predicate $marked(v)$ holds for objects that have been marked, and the unary predicate $pending(v)$ holds for objects that are pending to be marked. The binary predicates $left(v_1, v_2)$ and $right(v_1, v_2)$ hold for objects related by the `left` and `right` fields, respectively.

Note how the transitive-closure operator allows us to naturally express the notion of reachability from the root in these formulae. Also note that the existence of individuals has to be explicitly specified (via the ϵ existence predicate). This stems from the use of the constant-domain semantics, which explicitly expresses existence of individuals.

5.2.2 Trace Semantics

It is natural to consider the concrete semantics of a program as the set of its execution traces [26, 107], where each trace is an infinite sequence of *configurations*. First-order logical structures provide a natural representation of configurations with an unbounded number of objects: an individual of the structure's domain (universe) corresponds to an anonymous, unique store location, and predicates represent properties of store locations. Such a representation allows properties of the heap contents to be maintained while abstracting away any information about the actual physical locations in the store.

A *program configuration* encodes a global state of the program. Technically, we use a first-order logical structure to represent a program configuration.

Definition 5.2.3 (Constant-domain program configuration) *A **configuration** is represented via a first-order logical structure $C^{\natural} = \langle U_c, \iota_c \rangle$, where U_c is the infinite domain (universe) of the structure, and ι_c is the interpretation function that maps predicates to their truth values; that is, for each $p \in \mathcal{P}^k$, $\iota_c(p) \colon U_c^k \rightarrow \{0, 1\}$, such that for all $u \in U_c$, $\iota_c(eq)(u, u) = 1$, and for all $u_1, u_2 \in U_c$ such that u_1 and u_2 are distinct individuals $\iota_c(eq)(u_1, u_2) = 0$. We also require that $\iota_c(\epsilon)(u) = 1$ only for a finite number of individuals. (However, there is no a priori upper bound on the number of individuals u in a structure for which $\iota_c(\epsilon) = 1$.)*

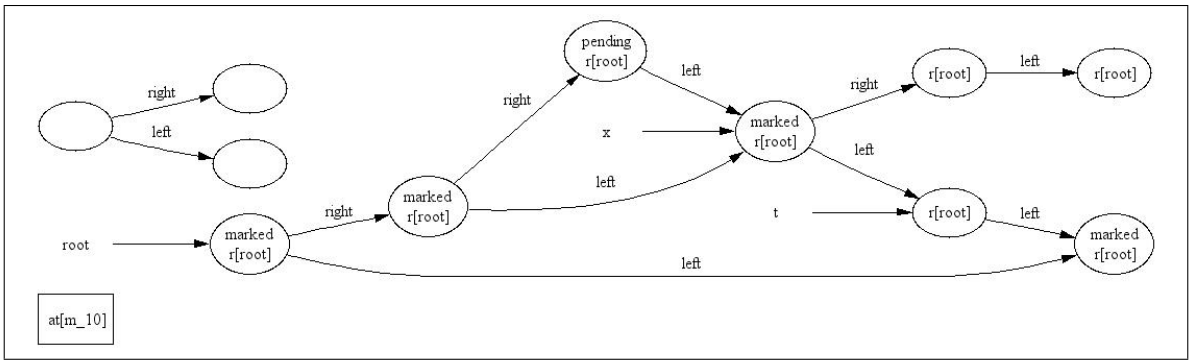


Figure 5.2: A possible configuration $C^{\natural}_{5.2}$ of the marking procedure.

As in previous chapters, we use directed graphs to depict configurations. Each individual of the universe that actually exists in the configuration is shown as a node. Individuals for which $\epsilon(v)$ is 0 are not shown. A unary predicate $p(v)$ that holds for an individual u is drawn inside the node u; the predicate $\epsilon(v)$ is not shown because it holds for all individuals depicted. A predicate that can hold at most for a single object (e.g., $x(v)$) is shown as an edge from the predicate symbol to the node in which it holds. A binary predicate $p(u_1, u_2)$ that evaluates to 1 is drawn as directed edge from u_1 to u_2 labeled with the predicate symbol. Finally, a nullary predicate $p()$ is drawn inside a box.

Example 5.2.4 *Fig. 5.2 shows a possible configuration of the marking procedure when execution is at program label m_{10}. In this configuration,* `root` *points to an object from which seven other objects are reachable by transitively traversing* `right` *and* `left` *fields. The predicate $r[root]$ records the value of the formula φ_r of Example 5.2.2, corresponding to transitive reachability from* `root`*. The importance of recording this value as a predicate will become clear in later sections. Three objects in this configuration have already been marked by previous loop iterations; these are the objects for which the predicate $marked$ holds. Similarly, two objects in the configuration are in the pending set (predicate $pending$ holds for these). In this configuration, three objects are not reachable from* `root` *and shown in the upper part of the figure. The nullary predicate $at[m_{10}]$ shown inside a box indicates that procedure execution is currently at label m_{10}.*

We define an operational semantics that manipulates configurations by means of *actions*. Informally, an action consists of a precondition under which the action is enabled and a definition of how predicates are updated as a result of the action. In this chapter, we prefer the use of two-vocabulary formulae for specifying actions (over the precondition/update formulation of Chapter 2) as two-vocabulary formulae naturally correspond to program transitions, and are also used to define progress measures in later sections. More formally, an action is defined as follows:

Definition 5.2.5 (Action) *An **action** is a **two-vocabulary formula** over $\mathcal{P}$, that is, a formula over the predicates in $\mathcal{P} \cup \{p' | p \in \mathcal{P}\}$ containing primed and non-primed versions of predicates in $\mathcal{P}$. An action*

*consists of two parts: (i) a **precondition**, containing only non-primed predicates; (ii) an update part which is allowed to include both non-primed and primed predicates. We say that an action is **enabled** in a configuration when its precondition has a satisfying assignment in that configuration. For brevity, we assume that a primed predicate not appearing in the formula maintains its original (non-primed) value.*

Example 5.2.6 *Given a vocabulary*
$\mathcal{P} = \{at[m_1](), \ldots, at[m_{17}](), x(v), t(v), left(v_1, v_2), right(v_1, v_2)\}$*, the action*

$$at[m_9]() \wedge \neg at[m_9]'() \wedge at[m_{10}]'() \wedge \forall v.(t'(v) \iff \exists v_1.x(v_1) \wedge left(v_1, v))$$

represents a transition in which program location changes from label m_9 to m_{10} and t is assigned the value of `x.left`*. Its precondition, $at[m_9]()$, is enabled at configurations that represent a state of program execution that is at label m_9.*

In this thesis, we assume that the program is represented as a first-order transition system. To simplify the presentation, we do not discuss the fairly standard additional Justice and Compassion conditions that are added to support the notions of weak and strong fairness (see [69, 81]).

In the sequel, we use the term program to also refer to the first-order transition system that represents the program.

Definition 5.2.7 (First Order Transition System (FOTS)) *A **first order transition system (FOTS)** is a pair $\langle \mathcal{P}, \tau \rangle$ where $\mathcal{P}$ is a vocabulary, and τ is the set of actions given as a set of two-vocabulary formulae.*

A first order transition system induces a transition relation as follows.

Definition 5.2.8 (Transition Relation) *Given a program, we say that a configuration $C^\natural$ **rewrites** into a configuration $C^{\natural\prime}$ under ac (denoted by $C^\natural \overset{ac}{\Rightarrow} C^{\natural\prime}$) where ac is a program action, if ac is enabled at $C^\natural$ and the values of predicates in $C^{\natural\prime}$ are determined by the primed predicates in ac. We write $C^\natural \Rightarrow C^{\natural\prime}$ when there exists an action ac s.t. $C^\natural \overset{ac}{\Rightarrow} C^{\natural\prime}$.*

We make the usual assumption that the transition relation is total. This allows us to only consider infinite traces. For terminating programs we make the transition relation total by letting the final configuration rewrite into itself with an "idle action".

Definition 5.2.9 (Trace) *Given a program, a **trace** is an infinite sequence of program configurations $C_0^\natural, C_1^\natural \ldots$, where each configuration represents a global state of the program, $C_0^\natural$ is an initial configuration, and for each $C_i^\natural$ and $C_{i+1}^\natural$, $C_i^\natural \Rightarrow C_{i+1}^\natural$.*

The semantics for ETL is a constant-domain semantics, in which all configurations of a program share the same (infinite) universe. We denote this universe by U and note that $U_{\pi_0} = \ldots = U_{\pi_i} = \ldots = U$. The quantification in the constant-domain semantics is a possibilist quantification that ranges over all individuals of U, that is, over individuals that *possibly exist* in a configuration. Individuals that *actually exist* in a configuration are distinguished by having the value 1 for the predicate ϵ (and ϵ can be used to write queries restricted to individuals that actually exist).

In the following, $head(\pi)$ denotes the first configuration in a trace π, $tail(\pi)$ denotes the suffix of π without the first configuration, and π^i denotes the suffix of π starting at the i-th configuration.

The goal of the ETL semantics we present is to find whether a violation-formula, i.e., a formula defining forbidden behaviors, could hold for a program. The following trace-based semantics is therefore an existential trace semantics in which a program is said to satisfy a given property if there exists a program trace that satisfies the property.

Throughout the chapter, we assume that formulae are in **positive normal form**, in which negations only appear over atomic formulae. For simplicity and brevity, we only show the semantics for future ETL (where no past operators are used). The semantics of the past operators is added in the standard manner as in [69], or in a manner similar to [26] (the interested reader can find the full definitions in Appendix C.3).

Definition 5.2.10 (ETL Trace-based Semantics) *We define when an ETL formula φ is satisfied over a trace π with an assignment Z (denoted by $\pi, Z \models \varphi$) as follows:*

- $\pi, Z \models \mathbf{1}$, *and not* $\pi, Z \models \mathbf{0}$.
- $\pi, Z \models p(v_1, \ldots, v_k)$ *when* $\iota_{head(\pi)}(p)(Z(v_1), \ldots, Z(v_k)) = 1$
- $\pi, Z \models \neg\varphi$ *when not* $\pi, Z \models \varphi$
- $\pi, Z \models \varphi \vee \psi$ *when* $\pi, Z \models \varphi$ *or* $\pi, Z \models \psi$
- $\pi, Z \models \exists v.\varphi(v)$ *when there exists* $u \in U$ *s.t.* $\pi, Z[v \mapsto u] \models \varphi(v)$
- $\pi, Z \models (TC\ v_1, v_2 \colon \varphi)(v_3, v_4)$ *when there exists* $u_1, \ldots, u_{n+1} \in U$, *s.t.* $Z(v_3) = u_1, Z(v_4) = u_{n+1}$, *and for all* $1 \leq i \leq n, \pi, Z[v_1 \mapsto u_i, v_2 \mapsto u_{i+1}] \models \varphi$.
- $\pi, Z \models \bigcirc\varphi$ *when* $tail(\pi), Z \models \varphi$.
- $\pi, Z \models \varphi \mathcal{U} \psi$ *when there exists* $k \geq 0$, *s.t.,* $\pi^k, Z \models \psi$ *and for all* $0 \leq j < k$, $\pi^j, Z \models \varphi$.
- $\pi, Z \models \varphi \mathcal{W} \psi$ *when there exists* $k \geq 0$, *s.t.,* $\pi^k, Z \models \psi$ *and for all* $0 \leq j < k$, $\pi^j, Z \models \varphi$, *or for all* $j \geq 0$, $\pi^j, Z \models \varphi$.

We omit definitions for $\wedge, \forall$ since they are defined similarly. We write $\pi \models \varphi$ when $\pi, Z \models \varphi$ for every assignment Z. Given a program P, we say that $P \models \varphi$ when there exists a trace π of the program P, such that $\pi \models \varphi$.

Note that the trace-based semantics guarantees that an eventuality (represented using the until operator) is fulfilled in a finite future.

5.2.3 State-Based Semantics

We now define a state-based semantics for ETL. The state-based semantics we define is an abstraction of the trace-based semantics of Definition 5.2.10 — we say that a state (configuration) existentially satisfies an ETL formula when there exists a trace emanating from the configuration (i.e., a suffix for future properties, and a prefix for past properties) that satisfies the ETL formula.

To allow the semantics to treat different assignments separately, we restrict our attention to ETL formulae that only allow a bounded number of simultaneous temporal requirements, i.e., we allow conjunctions, but forbid universal quantification over temporal subformulae. We still allow universal quantifiers to be used in a configuration-local manner, i.e., when the formula under the quantifier does not include temporal operators. Note that since the semantics is used to evaluate the violation formula, the subset we consider includes universally-quantified specifications.

We use the term *bounded demonic formula* to denote the fact that the number of requirements that should simultaneously hold for the formula is bounded (the term *demonic* is intentionally borrowed from demonic-nondeterminism).

Definition 5.2.11 (Bounded-Demonic ETL) *An ETL formula in positive normal form is a **bounded-demonic ETL formula** (BDETL) when no temporal operator appears under a universal quantifier.*

Definition 5.2.12 (ETL Existential State-Based Semantics) *Given a set of BDETL formulae F, and a program P, we say that F is **existentially satisfied** from a configuration (state) $C^\natural$ with an assignment Z (denoted by $C^\natural, Z \models_E F$) when one of the following conditions holds:*

(A0) $F = \emptyset$

(A1) $F = F' \cup \{\mathbf{1}\}$ *and* $C^\natural, Z \models_E F'$,

(A2) $F = F' \cup \{p(v_1, \ldots, v_k)\}$ *and* $\iota_{C^\natural}(p)(Z(v_1), \ldots, Z(v_k))$=1, *and* $C^\natural, Z \models_E F'$

(A3) $F = F' \cup \{\neg\varphi\}$ *and not* $C^\natural, Z \models_E \{\varphi\}$, *and* $C^\natural, Z \models_E F'$

(A4) $F = F' \cup \{\varphi \vee \psi\}$ *and* $C^\natural, Z \models_E F' \cup \{\varphi\}$ *or* $C^\natural, Z \models_E F' \cup \{\psi\}$

(A5) $F = F' \cup \{\varphi \wedge \psi\}$ *and* $C^\natural, Z \models_E F' \cup \{\varphi, \psi\}$

(A6) $F = F' \cup \{\exists v.\varphi(v)\}$ *and there exists* $u \in U_{C^\natural}$ *s.t.* $C^\natural, Z[v \mapsto u] \models_E F' \cup \{\varphi(v)\}$

(A7) $F = F' \cup \{\bigcirc\varphi\}$ *and exists* $C^{\natural\prime}$, $C^\natural \Rightarrow C^{\natural\prime}$ *s.t.,* $C^{\natural\prime}, Z \models_E \{\varphi\}$ *and* $C^\natural, Z \models_E F'$.

(A8) $F = F' \cup \{\varphi \mathcal{U} \psi\}$ *and* $C^\natural, Z \models_E F' \cup \{\psi\}$ *or*
$C^\natural, Z \models_E F' \cup \{\varphi\}$ *and there exists* $C^{\natural\prime}$ *s.t.* $C^\natural \Rightarrow C^{\natural\prime}$ *and* $C^{\natural\prime}, Z \models_E \{\varphi \mathcal{U} \psi\}$.

(A9) $F = F' \cup \{\varphi \mathcal{W} \psi\}$ *and* $C^\natural, Z \models_E F' \cup \{\psi\}$ *or*
$C^\natural, Z \models_E F' \cup \{\varphi\}$ *and there exists* $C^{\natural\prime}$ *s.t.* $C^\natural \Rightarrow C^{\natural\prime}$ *and* $C^{\natural\prime}, Z \models_E \{\varphi \mathcal{W} \psi\}$.

We omit the definition for $\forall$ since it is defined similarly. This semantics has the additional requirement that eventualities (i.e., $\varphi \mathcal{U} \psi$) are fulfilled within a finite future (note that this requirement is what differentiates (A8) from (A9) which otherwise have identical structure).

The rules of Definition 5.2.12 could be viewed as rewrite rules for the construction of a joint product tableau combining a program and a violation property. These rules could be then viewed as a generalization of the rules used in PTL tableau construction [69]. In this respect, note that formulae in the set F of Definition 5.2.12 are only formulae contained in the closure of the original property Φ, where the closure of Φ is defined to be all its subformulae and their negations.

The following theorem establishes a connection between the state-based semantics defined above and the trace-based semantics of Definition 5.2.10.

Theorem 5.2.13 *Given a BDETL formula* φ*, and a program* P, $P \models \varphi \implies P \models_E \{\varphi\}$

Proof: in Appendix B

Theorem 5.2.13 allows us to check whether a program P satisfies a BDETL violation-property φ by checking if its initial configuration satisfies the set $\{\varphi\}$ according to the semantics of Definition 5.2.12.

Note that it is still not obvious how to compute the state-based semantics (as was the case with the trace semantics) since it may require reasoning about infinite traces.

Reasoning about infinite traces is required to show that there exists a violation trace for a liveness property. In a finite-state system this can be done by compactly representing infinite traces as cycles in the finite state-space. In an infinite-state system, this is more complicated since an infinite trace does not necessarily repeat past configurations.

5.3 Abstract Semantics

Because the semantics of the previous section may be non-terminating, we now show how to guarantee termination by conservatively evaluating BDETL formulae using abstract interpretation [25].

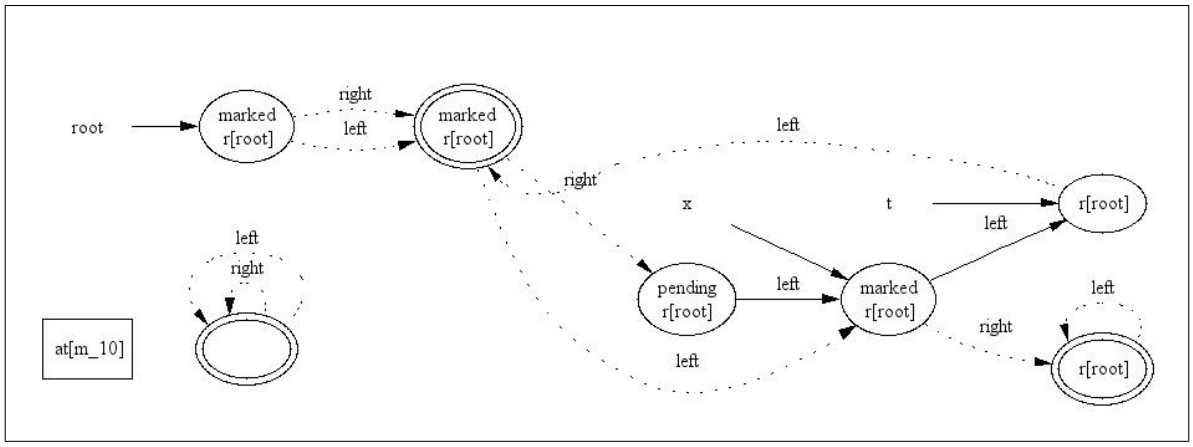


Figure 5.3: An abstract configuration $C_{5.2}$ that represents the concrete configuration $C^{\natural}_{5.2}$.

5.3.1 Abstract Configurations

We conservatively represent multiple concrete program configurations using a single logical structure with an extra truth-value $1/2$, which denotes values which may be 1 and may be 0. We allow an abstract configuration to include *summary nodes*, i.e., individuals that represent one or more individuals in a represented concrete configuration. Technically, a summary node u has $\iota(eq)(u,u) = 1/2$.

Definition 5.3.1 (Abstract configuration) *An **abstract configuration** is a 3-valued logical structure $C = \langle U, \iota \rangle$ where: (i) U is the universe of the structure, each individual in U represents possibly many objects; (ii) ι is the interpretation function mapping predicates to their truth-value in the structure, i.e., for every predicate $p \in P$ of arity k, $\iota(p) \colon U^k \to \{0, 1/2, 1\}$.*

As mentioned in Section 2.4.3, it is possible to refine the abstraction by adding *instrumentation predicates* [91]. In this chapter, we use the reachability instrumentation predicate of [91] for recording reachability from the `root` variable. The unary instrumentation predicate $r[root](v)$ holds for individuals that are reachable from the `root` variable.

Example 5.3.2 *Fig. 5.3 shows an abstract configuration $C_{5.2}$ that represents the concrete configuration $C^{\natural}_{5.2}$ of Fig. 5.2. Nodes with double-line boundaries are summary nodes, which possibly represent multiple concrete nodes. Dotted edges represent binary predicates with the value $1/2$. Note that $r[root](v)$ holds for all nodes excluding a single summary node that represents the "garbage" objects that are not reachable from `root`.*

Canonical Abstraction

As in Chapter 2, we use *canonical abstraction* to define how configurations are represented by abstract configurations. Canonical abstraction maps concrete individuals to an abstract individual based on the

values of the individuals' unary predicates. All individuals having the same values for unary predicate symbols are mapped by the abstraction to the same abstract individual. Canonical abstraction guarantees that the resulting abstract configuration is of bounded size. We denote the canonical abstraction of a configuration C by *blur*(C), and denote by $C \sqsubseteq_{blur} C'$ the fact that canonical abstraction *embed* C *into* C'.

5.3.2 Abstract Semantics

The abstraction of the previous section induces an abstract transition relation defined as follows.

Definition 5.3.3 (Abstract Transition Relation) *We say that an abstract configuration C* ***rewrites*** *into an abstract configuration C' under ac (denoted by $C \overset{ac}{\Rightarrow}_\sharp C'$) where ac is an action, if for C and for C' there* exists *$C^\natural$ and $C^{\natural\prime}$ such that: (i) $C^\natural$ is in the concretization of C, i.e., C* represents *$C^\natural$; (ii) ac is enabled at $C^\natural$ and $C^{\natural\prime}$ is a result of the updates applied by ac; (iii) canonical abstraction embeds $C^{\natural\prime}$ into C'. We write $C \Rightarrow_\sharp C'$ if for some action ac $C \overset{ac}{\Rightarrow}_\sharp C'$.*

Note that this abstract transition relation only adds behaviors over the concrete transition relation. In particular, for any two concrete configurations $C^\natural \Rightarrow C^{\natural\prime}$, the abstract transition relation relates their abstractions, i.e., *blur*$(C^\natural) \Rightarrow_\sharp$ *blur*$(C^{\natural\prime})$.

To define the abstract semantics, we first define the concretization of an abstract configuration and an assignment: $\gamma(\langle C, Z\rangle) = \{\langle C^{\natural\prime}, Z'\rangle \mid C^{\natural\prime} \sqsubseteq_{blur} C, blur \circ Z' = Z\}$

Definition 5.3.4 (Abstract State-Based Semantics) *The abstract state-based semantics is derived from Definition 5.2.12 by using abstract interpretation. We define when an abstract configuration C and an assignment Z* ***potentially satisfy*** *a set of BDETL formulae F, as follows: $C, Z \models^\sharp_E F$ when there exists $\langle C^{\natural\prime}, Z'\rangle \in \gamma(\langle C, Z\rangle)$ such that $C^{\natural\prime}, Z' \models_E F$*

The following theorem establishes the soundness of the abstract semantics. Intuitively, the abstract transition relation only adds behaviors over the concrete transition relation. Therefore, any behavior that was present in the concrete transition relation is also present in the abstract transition relation.

Theorem 5.3.5 *For a program P and a set of BDETL formulae F, $P \models_E F \Longrightarrow P \models^\sharp_E F$*

Proof: in Appendix B

Note that the above theorem does not guarantee completeness. Indeed, the abstract semantics may potentially satisfy formulae that are not satisfied by the concrete semantics.

Generally, the concretization of an abstract configuration and an assignment may be an infinite number of concrete configurations, and is therefore non-computable. In practice, our abstract semantics is

based on direct manipulation of abstract configurations (without the need to compute the concretization) which is more conservative [91].

Theorem 5.3.5 guarantees that if there exists a trace that satisfies a property φ under the concrete semantics, there also exists a trace that potentially satisfies φ under the abstract semantics. We use this theorem combined with Theorem 5.2.13 to check whether a given program potentially satisfies the violation-property φ.

If a program potentially satisfies the violation-property then it may have an incorrect behavior, but this behavior may also be a spurious behavior not exhibited by the concrete semantics and resulting from the abstraction. Otherwise, when the program does not satisfy the violation-property, it is guaranteed that the concrete semantics also exhibits only desired behaviors (i.e., behaviors that do not satisfy the violation property). The soundness of our approach is guaranteed by the following theorem

Theorem 5.3.6 *For a program P, and a BDETL formula φ, $P \models \varphi \Longrightarrow P \models^{\sharp}_{E} \varphi$*

Proof: $P \models \varphi \Longrightarrow P \models_E \{\varphi\}$ [Thm.5.2.13], $P \models_E \{\varphi\} \Longrightarrow P \models^{\sharp}_{E} \{\varphi\}$ [Thm.5.3.5]

The abstract semantics of Definition 5.3.4 requires that eventualities are fulfilled within a finite future. Since abstract configurations are of bounded size, the number of abstract configurations is finite, and the abstract transition relation is guaranteed to be finite and thus contain cycles. The requirement that eventualities are fulfilled within a finite future may be violated by some cycles.

It is important to note that we are trying to find a trace in which the violation-property holds. Thus, we may conservatively say that the violation-property holds even when it doesn't hold in the concrete semantics. Therefore, saying that the violation-property holds due to a spurious cycle is sound.

5.4 First-Order Representation

In this section, we use first-order logic with transitive closure to encode the existential state-based semantics of the previous section. To show how the abstract existential state-based semantics of Definition 5.3.4 is expressed in terms of a first-order transition system, we first show how to express the concrete existential state-based semantics of Definition 5.2.12 as a first-order transition system, and then apply the abstraction of Section 5.3.2, which produces a 3-valued first-order transition system for the semantics of Definition 5.3.4.

5.4.1 ETL Existential State-Based Semantics as First-Order Transition System

Given an FOTS representing the program to be verified, and a BDETL property, we construct an augmented FOTS that encodes the semantics of Definition 5.2.12 and combines it with the program's FOTS. The combined FOTS could be viewed as performing an interpretation of the ETL formula where the underlying program states are constructed on-the-fly as formula evaluation takes place.

We augment the vocabulary of the original transition system with a vocabulary for encoding the semantics. We note that the elements of the sets used while evaluating a formula in the existential state-based semantics can only be formulae in the closure of the formula (subformulae of the original formula, and their negations).

We use predicates corresponding to the formulae in the closure of the ETL formula as our vocabulary. We call predicates in this vocabulary **formula-predicates**. These predicates model members of the set F of Definition 5.2.12: a formula-predicate has the value 1 when the formula is in the set F. Intuitively, these predicates correspond to future obligations that should be satisfied by the program's execution. We denote by $\langle\varphi\rangle$ the predicate recording the fact that φ should be satisfied by the future (or present) of program execution. Initially, we require that the future of the execution satisfies the violation property. As the analysis progresses, future obligations may be fulfilled, possibly leading to a repeatable state in which no further obligations exist, satisfying the violation property.

The transitions in the FOTS are directly derived from the rules in Definition 5.2.12. The FOTS consists of two types of transitions: (i) semantic transitions, which correspond to the way the set F is updated by the rules of the semantics. These transitions update the formula-predicates and could be viewed as tableau rewrite rules; (ii) program transitions, which correspond to state updates as result of a step taken by the program. These transitions update predicates of the original vocabulary (rather than formula-predicates).

Note that when the semantics uses a program transition to find a successor state, it chooses from all enabled actions. Technically, the transition $C^{\natural} \Rightarrow C^{\natural\prime}$ is realized as $\bigvee_i ac_i$ where ac_i are the actions of the program (as two-vocabulary formulae).

Evaluation of an ETL formula using the semantics of Definition 5.2.12 corresponds to a run of the first-order transition system encoding the semantics of the formula.

Definition 5.4.1 (Run of an FOTS) *A **run** of an FOTS is an infinite sequence of first-order configurations $C_0^{\natural}, C_1^{\natural}, \ldots$ where $C_0^{\natural}$ is an initial state, and each configuration $C_{i+1}^{\natural}$ is derived from its predecessor $C_i^{\natural}$ by a single action of the transition system.*

Recall that the semantics of Definition 5.2.12 requires that eventualities are satisfied within a finite future. To express this requirement, we distinguish between formula-predicates that are allowed to be repeated forever, and formulae-predicates that represent formulae that are required to be satisfied within a finite-future. A formula-predicate is said to be *accepting* if its formula is of the form $\varphi \mathcal{W} \psi$ or is 1. All non-accepting formula-predicates represent subformulae that are required to be satisfied within a *finite future*.

Definition 5.4.2 (Acceptance) *A run of an FOTS is **accepting** when every non-accepting formula-predicate gets the value 0 within a finite future. An FOTS is said to be **accepting** when it has an accepting run.*

The following definition requires that the translation from the property to an FOTS preserves correctness, that is, that the FOTS constructed for a property φ is accepting if and only if the program satisfies φ.

Definition 5.4.3 *Given a BDETL formula φ, let T_φ be the FOTS constructed for evaluating φ. We say that the translation is correct when a program P and an initial configuration c satisfy φ if and only if T_φ is accepting when starting from c.*

The following example shows how an FOTS is constructed for the example property $(P1)$.

Example 5.4.4 *Consider the property $\Phi = \forall^{\epsilon} v.r[root](v) \rightarrow \Diamond\, marked(v)$. We start by taking the negation of the property $\varphi = \neg\Phi = \exists^{\epsilon} v.r[root](v) \wedge \Box\, \neg marked(v)$, and taking its closure. Using $\langle\psi\rangle(x_1, \ldots, x_k)$ to denote the formula-predicate for a formula ψ with free variables $x_1, \ldots, x_k$, we define the predicates*

$$\{\langle \exists^{\epsilon} v.r[root](v) \wedge (\neg marked(v)\, \mathcal{W} 0)\rangle(), \langle r[root](v) \wedge (\neg marked(v)\, \mathcal{W} 0)\rangle(v),$$
$$\langle (\neg marked(v)\, \mathcal{W} 0)\rangle(v), \langle \epsilon(v)\rangle(v), \langle marked(v)\rangle(v), \langle r[root](v)\rangle(v)\}$$

and also their version for the next program-successor, which we denote by $\langle\varphi\rangle^{\bullet}$ (recall that these predicates correspond to future obligations that should be satisfied by the program's execution). We denote by $\langle\varphi\rangle'$ the value of the predicate in the next configuration of semantics evaluation, that is, after a semantics transition. Note that $\langle\varphi\rangle^{\bullet}$ denotes the value of the predicate after a program transition.

The semantics transitions are shown in Table 5.2. For example, the encoding of the rule $(A6)$ corresponds to the expansion of an existential quantifier. In this transition, the nullary formula predicate $\langle \exists^{\epsilon} v.r[root](v) \wedge (\neg marked(v)\, \mathcal{W} 0)\rangle()$ is expanded to a non-deterministic selection of an individual for which the unary formula predicate $\langle r[root](v) \wedge (\neg marked(v)\, \mathcal{W} 0)\rangle$ holds. Fig. 5.4 shows a single successor (out of the many possible successors) resulting from the application of the rule $(A6)$ to an initial concrete configuration of the marking procedure.

Abstract Semantics

The above definitions and Definition 5.4.3 show how to implement the concrete existential state-based semantics as a first-order transition system. Because the implementation is given in terms of a first-order transition system, we can apply the abstraction of Section 5.3.2 to perform an abstract interpretation of this transition system.

The acceptance condition of Definition 5.4.2 is realized in terms of repeated reachability of abstract configurations. This may lead to spurious acceptance when a repeated abstract configuration represents an infinite sequence of different concrete configurations. However, because we are trying to verify a violation-property, spurious acceptance is still a sound result.

Rule	**Instance**
$(A6)$	$\langle \exists^{\epsilon} v.r[root](v) \wedge (\neg marked(v)\, \mathcal{W} 0)\rangle()$ $\wedge \neg\langle \exists^{\epsilon} v.r[root](v) \wedge (\neg marked(v)\, \mathcal{W} 0)\rangle'() \wedge \exists^{\epsilon} v.\langle r[root](v) \wedge (\neg marked(v)\, \mathcal{W} 0)\rangle'(v)$
$(A5)$	$\langle r[root](v) \wedge (\neg marked(v)\, \mathcal{W} 0)\rangle(v)$ $\wedge \neg\langle r[root](v) \wedge (\neg marked(v)\, \mathcal{W} 0)\rangle'(v) \wedge \langle r[root](v)\rangle'(v) \wedge \langle \neg marked(v)\, \mathcal{W} 0)\rangle'(v)$
$(A2)$	$\langle r[root](v)\rangle(v) \wedge r[root](v) \wedge \neg\langle r[root](v)\rangle'(v)$ $\langle r[root](v)\rangle(v) \wedge \neg r[root](v) \wedge \neg\langle r[root](v)\rangle'(v) \wedge \langle 0\rangle'(v)$
$(A9)$	$\langle \neg marked(v)\, \mathcal{W} 0\rangle(v)$ $\wedge \neg\langle \neg marked(v)\, \mathcal{W} 0)\rangle'(v) \wedge \langle \neg marked(v)\rangle(v) \wedge \langle \neg marked(v)\, \mathcal{W} 0\rangle^{\bullet}(v)$
$(A2)$	$\langle \neg marked(v)\rangle(v) \wedge marked(v) \wedge \neg\langle \neg marked(v)\rangle'(v) \wedge \langle 0\rangle'(v)$ $\langle \neg marked(v)\rangle(v) \wedge \neg marked(v) \wedge \neg\langle \neg marked(v)\rangle'(v)$

Table 5.2: Transitions in the FOTS for the property of Example 5.4.4.

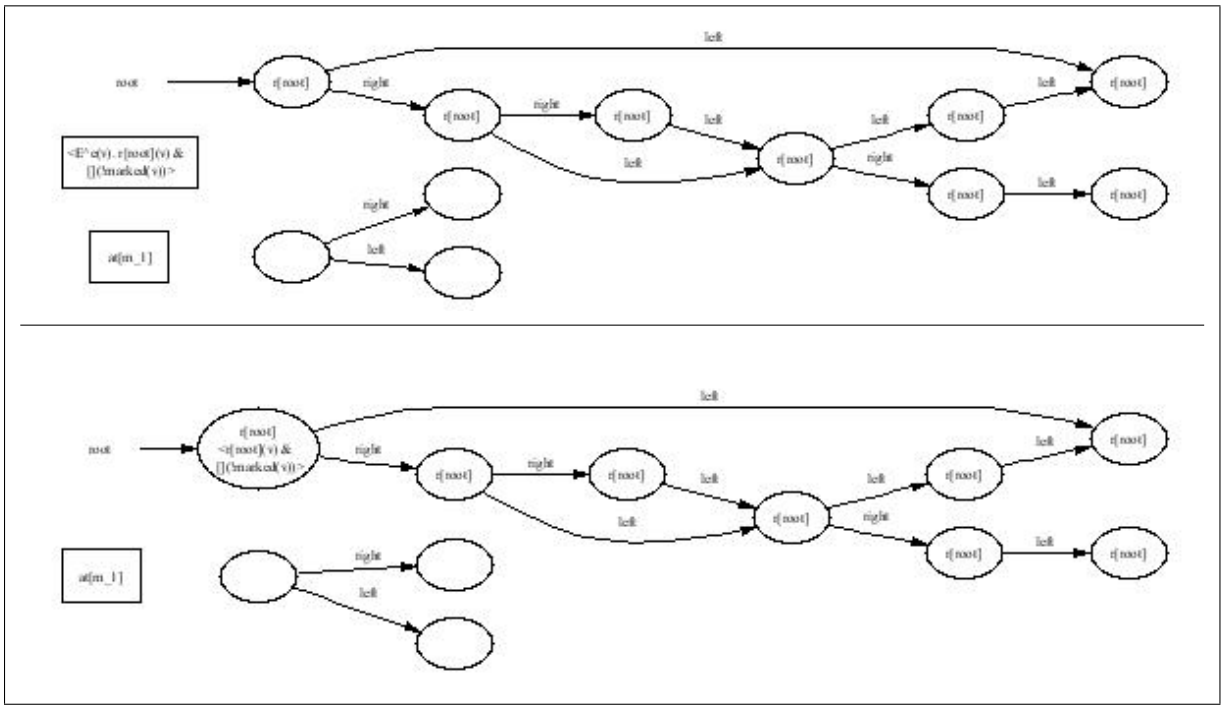


Figure 5.4: One successor derived by application of rule (A6) to an initial configuration.

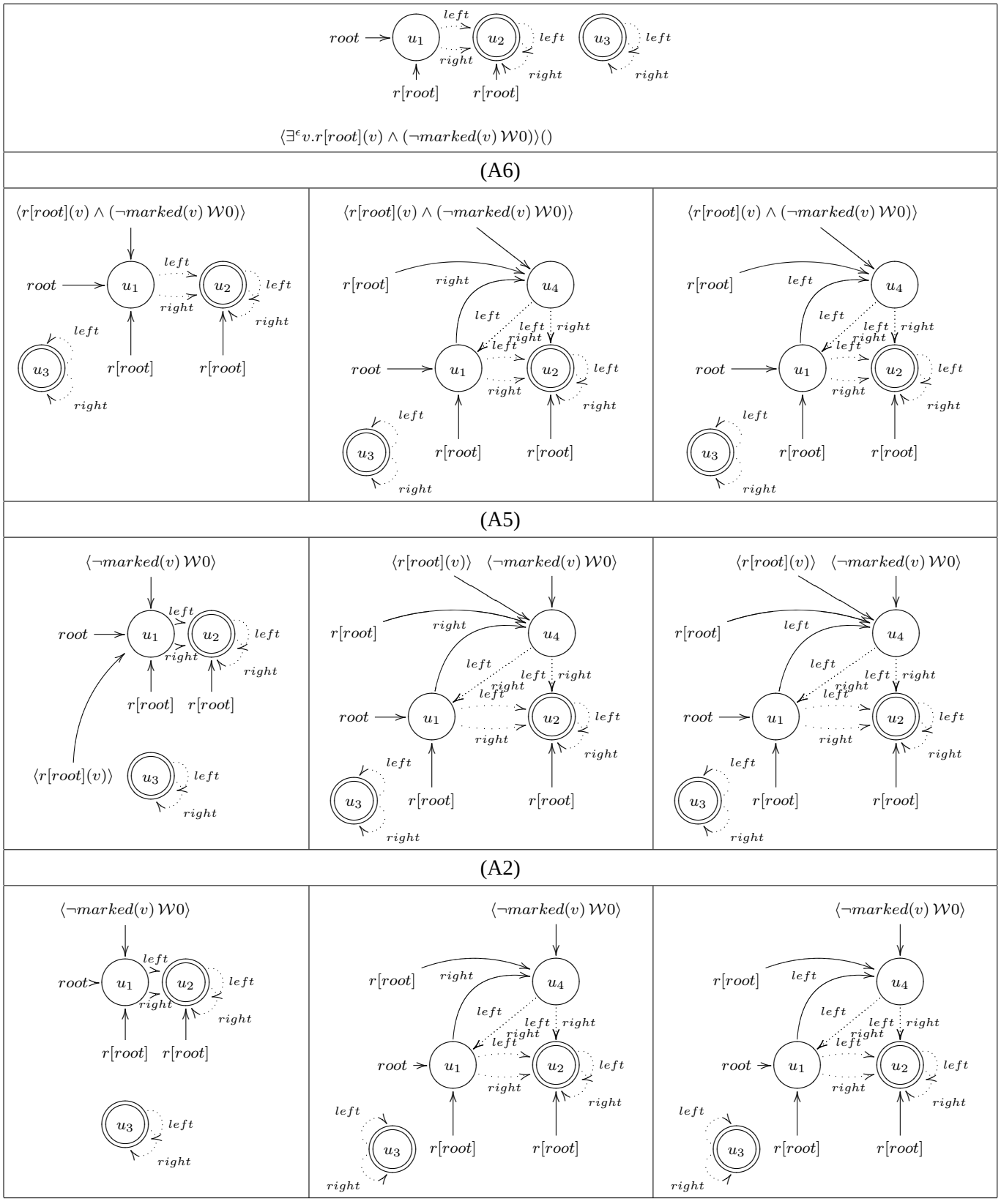


Figure 5.5: Partial abstract interpretation of the example FOTS. Only part of the abstract configurations are shown. Interpretation is continued on Fig. 5.6

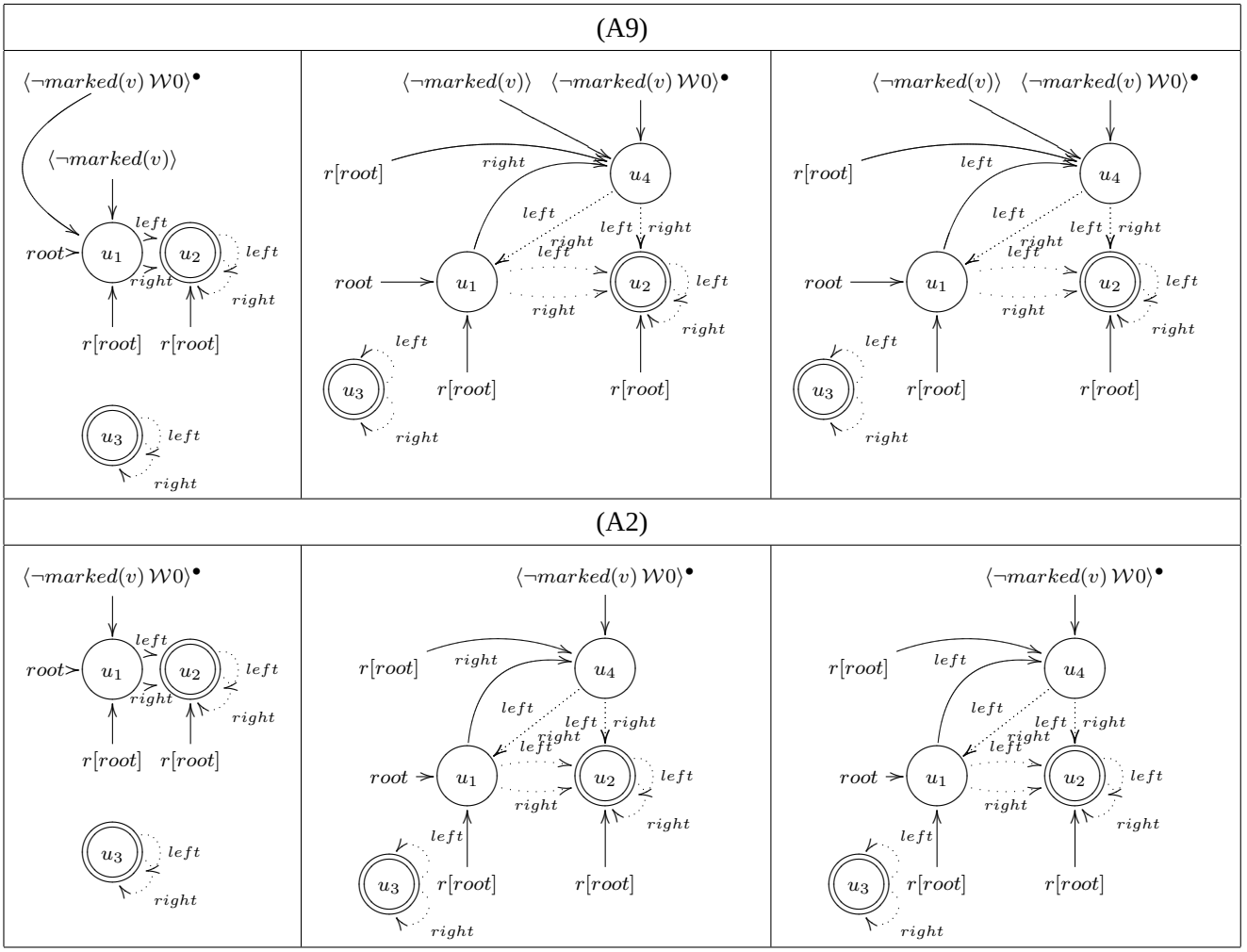


Figure 5.6: Partial abstract interpretation of the example FOTS, continued from Fig. 5.5. Only part of the abstract configurations are shown.

Example 5.4.5 *Fig. 5.5 and Fig. 5.6 show a partial abstract interpretation of the FOTS constructed for the violation property of the property* $(P1)$ *(as described in Example 5.4.4), and a possible initial abstract configuration at the entry to the marking procedure.*

Initially, the predicate $\langle \exists^{\epsilon} v.r[root](v) \wedge (\neg marked(v)\, \mathcal{W} 0)\rangle()$ *holds, corresponding to the fact that the violation property has to be accepted. In the first step, the rule* $(A6)$ *is applied to the initial abstract configuration, resulting in a set of abstract configurations (only* 3 *shown). In each of these resulting abstract configurations, a single individual is chosen as the individual for which the property is expected to hold. Following the transitions of the FOTS, rule* $(A5)$ *is now applicable, transforming the single predicate* $\langle r[root](v) \wedge (\neg marked(v)\, \mathcal{W} 0)\rangle$ *that should hold for an individual into two separate predicates* $\langle r[root](v)\rangle$ *and* $\langle \neg marked(v)\, \mathcal{W} 0\rangle$ *that should hold for the same individual. Next, we apply rule* $(A2)$ *that evaluates the non-temporal requirement of* $\langle r[root](v)\rangle$ *in the current abstract configuration and sets* $\langle r[root](v)\rangle$ *to* 0 *for the individuals for which* $\langle r[root](v)\rangle$ *should hold and which are reachable from the root (i.e., for which* $r[root](v)$ *holds). In each of the resulting abstract configurations, there is now a single individual for which the formula predicate* $\langle \neg marked(v)\, \mathcal{W} 0\rangle$ *holds. This expresses a requirement on the current configuration and future configurations, applying rule* $(A9)$ *transforms the requirement into two separate requirements: one for the current configuration, i.e.,* $\langle \neg marked(v)\rangle$*; and one for future configurations, i.e.,* $\langle \neg marked(v)\, \mathcal{W} 0\rangle^{\bullet}$*. Rule* $(A2)$ *again resolves the formula predicates that are satisfied by the current configuration.*

The resulting abstract configurations are configurations in which the only requirement is $\langle \neg marked(v)\, \mathcal{W} 0\rangle^{\bullet}$ *for the selected individual. The next step in the abstract interpretation of the FOTS is to evaluate a program transition, and turn* $\langle \neg marked(v)\, \mathcal{W} 0\rangle^{\bullet}$ *to* $\langle \neg marked(v)\, \mathcal{W} 0\rangle$ *in the resulting abstract configurations (as* $\langle \neg marked(v)\, \mathcal{W} 0\rangle^{\bullet}$ *expresses the requirement after a program's transition).*

Eventually, all reachable nodes will be marked, and requirements of the form $\langle \neg marked(v)\, \mathcal{W} 0\rangle$ *are therefore violated. Hence, the property violation property is not satisfied by the example program (and the original property holds).*

5.4.2 Liveness and Progress

Verifying liveness properties requires observing progress. To observe progress under abstraction, we need to consider the abstraction of transitions (pairs of configurations) rather than the abstraction of single configurations.

In order to observe progress, we classify transitions using a progress measure as suggested in [68, 75, 57], i.e., as being "helpful", "neutral" or "harmful" with respect to a predefined progress measure. A "helpful" transition is one that decreases the measure, a "neutral" transition does not change the progress measure, and a "harmful" transition increases the progress measure. Using this transition classification allows us to rule out abstract traces for which the progress measure is infinitely decreasing.

The progress measure we use for the running example is based on the fact that the set of individuals

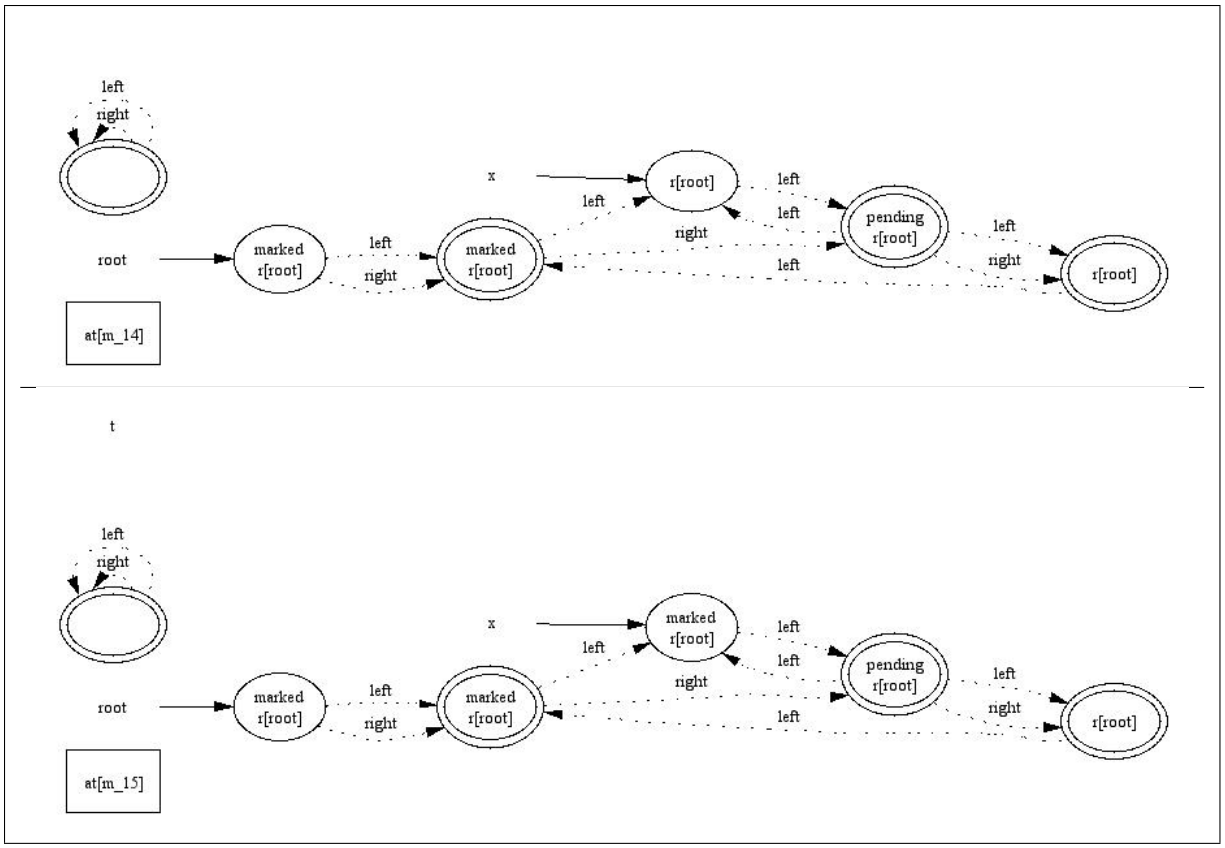


Figure 5.7: Abstract helpful transition from m_8 to m_9.

which are reachable from root and which are unmarked is decreasing on each iteration of the loop. We formulate the progress measure as the following two-vocabulary formulae:

$$\varphi_{\downarrow} = \exists^{\epsilon} v. \neg marked(v) \wedge r[root](v) \wedge (marked'(v) \vee \neg r[root]'(v))$$
$$\varphi_{\uparrow} = \exists^{\epsilon} v. (marked(v) \vee \neg r[root](v)) \wedge \neg marked'(v) \wedge r[root]'(v)$$

where $\varphi_{\downarrow}$ corresponds to a transition in which the progress measure is decreasing, and $\varphi_{\uparrow}$ to a transition in which the progress measure is increasing. A transition for which neither $\varphi_{\downarrow}$ nor $\varphi_{\uparrow}$ holds is a neutral transition that does not change the progress measure.

The above progress measure rules out the trace in which the loop does not terminate because the progress measure is infinitely decreasing along this trace.

Example 5.4.6 *The transition from m_9 to m_9 is a helpful transition in which a previously reachable but unmarked node becomes marked. This is shown in Fig. 5.7. Note that $\varphi_{\downarrow}$ holds for this transition. Also note that $\varphi_{\uparrow}$ does not hold for any transition of the example program.*

5.4.3 Safety Properties

For safety properties, verification could be performed more efficiently, without splitting successor configurations for different assignments. The intuition is that we only need to observe events in the past. This allows evaluating quantifiers in a configuration-local manner. In particular, formulating the properties of [95] and [115] using past formulae in our framework, yields the same algorithms as used there.

5.5 Conclusion

We have presented a framework for verifying temporal properties of sequential and concurrent heap-manipulating programs. The framework can be used for verification of safety and liveness properties. Appendix C provides additional ETL properties for the mark and sweep algorithm, and additional examples of ETL specifications.

The framework described in this chapter generalizes our previous work on verification of heap-manipulating programs and allows systematic formulation of properties that previously required ad-hoc solutions. In particular, it generalizes our work on verifying strong safety properties (Chapter 2), verifying non-nested liveness properties (Chapter 4), and verifying local temporal safety properties (Section 7.1).

Chapter 6

Verifying Safety Properties using Separation and Heterogeneous Abstraction

In this chapter, we show how *separation* (decomposing a verification problem into a collection of verification subproblems) can be used to improve the efficiency and precision of verification of safety properties. We present a simple language for specifying *separation strategies* for decomposing a single verification problem into a set of subproblems. (The strategy specification is distinct from the safety property specification and is specified separately.) We present a general framework of *heterogeneous abstraction* that allows different parts of the heap to be abstracted using different degrees of precision at different points during the analysis. We show how the goals of separation (i.e., more efficient verification) can be realized by first using a separation strategy to transform (instrument) a verification problem instance (consisting of a safety property specification and an input program), and by then utilizing heterogeneous abstraction during the verification of the transformed verification problem.

Some tasks are best done by machine,
while others are best done by human insight;
and a properly designed system will find the right balance.
– D. Knuth

6.1 Introduction

Recently there has been significant and growing interest in static verification of safety properties (e.g., see [21, 29, 6, 44, 43, 4, 84, 41, 27]). Such verification is valuable since it can identify software defects early on, thereby improving programmer productivity, reducing software development costs, and

```
 ...
10  ConnectionManager cm = new ConnectionManager();
11  Connection con1 = cm.getConnection();
12  Statement stmt1 = cm.createStatement(con1);
...
15  ResultSet maxRs = stmt1.executeQuery(maxQry);
16  if (maxRs.next())
...
18  ResultSet rs1 = stmt1.executeQuery(balancesQry);
19  if (maxBalance1 < threshold) {
20    stmt1.close();
21    closed1 = true;
22  }
23  Connection con2 = cm.getConnection();
24  Statement stmt2 = cm.createStatement(con2);
...
27  ResultSet rs2 = stmt2.executeQuery(balancesQry);
28  ResultSet maxRs2 = stmt2.executeQuery(maxQry);
29  if (maxRs2.next())
...
31  ResultSet minRs2 = stmt2.executeQuery(minQry);
...
40      while (rs2.next())
...
```

Figure 6.1: JDBC example snippet.

increasing software quality and reliability.

Consider the Java program fragment shown in Fig. 6.1. This program performs a number of database queries using JDBC [109]. This example violates one of the usage constraints imposed by the JDBC library. Specifically, the execution of a query in line 28, using a `Statement` object, has the implicit effect of discarding the results to the previous query executed in line 27 (using the same `Statement` object). Hence, the subsequent attempt to use these discarded results, in line 40, is invalid.

We are interested in verifying that a given program satisfies safety properties of the kind illustrated above. While significant progress has been made recently in such lightweight verification, doing precise verification that can scale to large and complex programs still remains a challenge. In this chapter, we investigate a technique to improve the precision and efficiency of such verification.

The starting point for our work is the notion of *separation*: the idea that separating or decomposing a verification problem into a collection of smaller subproblems can help scale verification algorithms (e.g., see [27]). Consider again the example in Fig. 6.1. This example program executes 5 different queries, producing 5 different `ResultSet`s. We can verify that the program satisfies the desired safety property by *independently* verifying the property for each of these `ResultSet`s.

It may seem like we are just restating the problem, but this restatement is important from the point of view of the underlying analysis. It can significantly increase the efficiency of the analysis by reducing the size of the state-space that needs to be explored. In our running example, `Statement stmt1` and `ResultSet rs1` can be in several possible states in line 28. While this information is relevant for verifying subsequent use of `ResultSet rs1`, it is irrelevant for verifying the usage of `ResultSet rs2`, for example. The motivation for separation is to exploit this to improve efficiency, without losing precision.

In this chapter, we explore this approach by addressing the following questions:

(1) How do we decompose a verification problem into a collection of subproblems?

(2) How can we adapt the state abstractions to each subproblem (so that we may achieve the desired

efficiency improvement)? One of the key characteristics of our approach is that we break up this question into two parts: (a) What are the objects that are *relevant* to a verification subproblem? (b) Given the set of relevant objects, how can we *adapt* the state abstraction to utilize this information?

In this chapter, we introduce the notion of a *separation strategy* as something that can help answer question (1) and partly help answer (2)(a). Rather than adopt a fixed strategy for separation, we introduce a simple language for specifying separation strategies that can be used to manually specify strategies. One strategy for the JDBC problem would be to apply separation at the level of a `Connection`, where verification of all `ResultSet`s created over a single `Connection` is treated as a single verification subproblem.

Currently, we see the strategy specification language as a way for analysis designers, such as ourselves, to specify and experiment with different strategies. Our intuition, however, is that end users may be able to easily identify objects of interest and relevance to some verification subproblem and that the strategy specification may be a lightweight way to allow end user input to guide verification.

Given a verification problem instance (consisting of a safety property specification and an input program) and a separation strategy, the first step of our approach is to *transform* (or instrument) the verification problem instance to reflect the separation strategy. (Here, it is worth pointing out that when we talk about "decomposing a verification problem into subproblems", we are talking at a conceptual level; the transformed verification problem mentioned above is equivalent to solving the subproblems in parallel.)

The second step is to perform verification for the transformed program and safety property in a way that exploits the separation. This leads us to question (2) above. One of the distinguishing characteristic of our approach is that we rely on an *integrated* analysis that performs, for example, heap analysis in conjunction with the verification (as opposed to performing it as a separate preceding analysis). Thus, we are interested in exploiting *separation* even for the heap analysis. (Indeed, the benefits of separation may be greatest for the heap analysis component if the verification utilizes precise, but expensive, heap analysis.)

In this chapter, we utilize *heterogeneous abstractions* that allow us to model different parts of the heap with different degrees of precision at different points in time as a technique to exploit separation.

Consider the example in Fig. 6.1. Fig. 6.2(a) informally shows two possible states of the heap at line `28`, corresponding to different branches taken at line `19`. The `Statement` referenced by `stmt1` and the `ResultSet` references by `rs1` are in a *closed* state in C_2 (as illustrated by the "c" inside the component node). Fig. 6.2(b) illustrates the abstract representation produced by our technique (with a simple separation): the representation above the line corresponds to one subproblem (corresponding to `Connection con1`), and the representation below the line corresponds to a different subproblem (corresponding to `Connection con2`). We present more details about these representations in later sections.

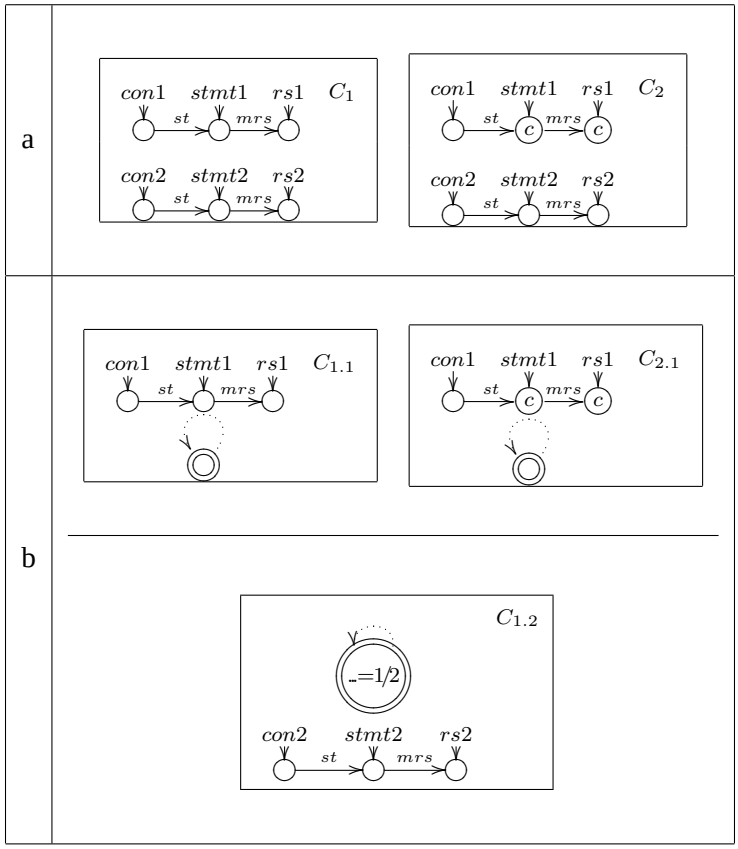


Figure 6.2: Separation and heterogenous abstraction.

Main Results

The main contributions of this chapter are:

- We present a simple language for specifying separation strategies for decomposing a single verification problem into a set of subproblems.
- We present a general framework of *heterogeneous abstractions* that allows different parts of the heap to be abstracted using different degrees of precision at different points during the analysis.
- We show how the goals of separation (i.e., more efficient verification) can be realized by first using a separation strategy to transform (instrument) a verification problem instance (consisting of a safety property specification and an input program), and then utilizing heterogeneous abstraction during the verification of the transformed verification problem.
- We have implemented a prototype of a separation verification engine using TVLA, and applied it to verify properties of several Java programs, using several different separation strategies. Initial results indicate that separation does improve the efficiency, and possibly precision, of verification results.

One of the themes to emerge in recent work (e.g., see [84, 41, 27]) is that maintaining just the right correlation required between "analysis facts" can be the key to efficient and precise verification: maintaining no correlations (independent attribute analysis) can lead to imprecision, while maintaining all correlations (relational analysis) can lead to inefficiency. However, finding this intermediate ground can be hard for heap analyses that, for instance, use graph-based representations of the heap. Our approach may be seen as a step towards achieving such a balance in a heap representation.

Existing approaches to verification range from more automated techniques that rely on no extra human input (other than the safety property specification) to techniques that rely on end users to provide significant annotation, such as program invariants. We see the strategy specifications we use as a potentially useful, lightweight, way for users to assist a verifier.

Related Work

ESP [27] is a system for typestate verification [103] that utilizes a simple fixed separation technique. Our work differs from ESP in several respects. ESP uses a two-phased approach to verification in which pointer-analysis is performed first, followed by typestate verification. This often prevents ESP from applying "strong" updates necessary for successful verification. Separation in ESP is exploited only in the typestate verification phase. We utilize an integrated analysis, where heap analysis and verification are performed simultaneously, allowing the heap analysis to benefit from separation. We also explore separation in a more general setting than ESP: we explore its applicability to first-order safety properties,

```
while (?)  {
   f = new File();
   f.read();
   f.close();
}
```

Figure 6.3: Program illustrating the difficulty of verifying that a file component is never read after it has been closed.

such as the ones shown earlier for JDBC, which involve relationships among multiple objects; we allow user-specifiable separation strategies; finally, our technique can achieve separation between multiple objects allocated at the same allocation site. Since our analysis is capable of separating out a *single* object (even from among multiple objects allocated at the same allocation site), it can apply "strong" updates when ESP is forced to use "weak" updates. This can lead to more precise results, as illustrated by the example in Fig. 6.3. Unlike ESP, our technique can successfully verify this example.

The instrumentation technique we use to implement separation strategies may be seen as an extension of techniques previously used (e.g., by Bandera [21, 22] and SLAM [72]) to instrument a program with respect to a safety property specification prior to verification. However, these approaches use such instrumentation purely to encode the verification problem, and do not exploit it for separation and the generation of adaptive abstractions as we do.

Separation is similar in spirit to McMillan's functional decomposition [70], which divides the verification task according to units-of-work rather than dividing it according to the program syntax. His division, however, is applied at the specification level since all entities have static names.

Guyer and Lin [50] show that it is valuable to have pointer analyses that are client-driven. His analysis is a two-pass analysis, with a client-independent first pass pointer analysis, followed by a second pass pointer analysis that uses different levels of context-sensitivity for different analyzed procedures, based on sources of imprecision identified using the results computed by the first pass.

[84] explores techniques to derive abstractions that are specialized to a safety property. Our work on separation is orthogonal to these techniques. In [95], a heap-safety-automaton (HSA) is used to specify local heap properties (corresponding to typestate properties) which are later verified without using any form of separation. We believe that the separation techniques in this chapter could be beneficial for their analysis as well.

Our heterogeneous abstraction technique is based on the parametric analysis framework of Sagiv et al. [91]. This analysis framework has been used to derive several powerful and precise, but very expensive, heap analyses. We believe that successful verification systems need to use such powerful analyses when needed (to handle difficult cases when they arise), but scalability requires that the scope

of such analyses be restricted to a small enough universe. We believe that the identification of "relevant" objects via our separation technique is a step towards achieving this.

An alternative separation technique would be to decompose a verification problem into subproblems that verify that each *use* of an object, such as a `ResultSet`, is safe, utilizing demand-driven analysis to solve the subproblems. This inherently involves "backward analysis", while our approach utilizes "forward analysis". The motivation for our approach is that "backward analysis" is inherently hard when complex heap analysis is involved.

6.2 Safety Properties

We are interested in verifying that client programs that use a component (library) satisfy correct usage constraints imposed by the library API. In this chapter, we use some of the usage constraints imposed by the JDBC library to illustrate our separation technique for verification of such safety properties.

The JDBC library allows client programs to create `Connection`s to databases. Any number of `Statement`s may be created over a `Connection`. A `Statement` can be used to execute a SQL query over the database, via the `executeQuery()` method, which returns the results of the query as a `ResultSet`. The `next()` method of a `ResultSet` can be used repeatedly to iterate over the results of the query. However, the execution of the `executeQuery()` method of a `Statement` implicitly *closes* any `ResultSet` previously returned by the `Statement`, and it is invalid to use any of those `ResultSet`s anymore. Similarly, after closing a `Connection`, it is invalid to use any of the `Statement`s created from that `Connection` or any of the `ResultSet`s returned by these `Statement`s.

Thus, the execution of line 28 in the example of Fig. 6.1 implicitly closes the `ResultSet` created in line 27, and this will cause an error when this closed `ResultSet` is used in line 40.

We specify safety properties using `Easl` [84], a procedural language for specifying an abstract semantics for a component library. `Easl` statements are a subset of Java statements containing assignments, conditionals, looping constructs, and object allocation. `Easl` types are restricted to booleans, heap-references, and built-in abstract Set and Map types. Finally, `Easl` provides a `requires` statement to specify the correct usage constraints imposed by the library: it is the responsibility of any program that uses the library to ensure that the condition specified by the `requires` clause holds at the corresponding program point. These are the safety properties we are interested in checking.

`Easl` supports object references and dynamic allocation. This allows us to naturally express the structural relationships between the objects of interest, as well as dynamic allocation of these objects.

Fig. 6.4 shows an `Easl` specification for the JDBC[1] safety properties described above.

Note the use of the set `statements` and the fields `myResultSet`, `myConnection`, and `ownerStmt`

[1] Field names from Sun's SDK1.3.1 sun.jdbc.odbc implementation.

```
class Connection {
 boolean closed;
 Easl.Set statements;
 Connection() {
  closed = false;
  statements = {};
 }
 Statement createStatement() {
  requires !closed;
  Statement st = new Statement(this);
  statements = statements U { st };
  return st;
 }
 void close() {
  closed = true;
  for each st in statements
   if (st.myResultSet != null) {
    st.closed = true;
    st.myResultSet.closed = true;
   }
 }
}
```

```
class Statement {
 boolean closed;
 ResultSet myResultSet;
 Connection myConnection;
 Statement(Connection c) {
  closed = false;
  myConnection = c;
  myResultSet = null;
 }
 ResultSet executeQuery(String qry) {
  requires !closed;
  if (myResultSet != null)
   myResultSet.closed = true;
  myResultSet = new ResultSet(this);
  return myResultSet;
 }
 void close() {
  closed = true;
  if (myResultSet != null)
   myResultSet.closed = true;
 }
}
```

```
class ResultSet {
 boolean closed;
 Statement ownerStmt;
 ResultSet(Statement s) {
  closed = false ;
  ownerStmt = s;
 }
 void close() {
  closed = true;
 }
 boolean next() {
  requires !closed;
 }
}
```

Figure 6.4: An `Easl` specification for a simplified subset of the JDBC API.

to specify the relationships between the components. Also note that applying `executeQuery` closes the `ResultSet` component referenced by `myResultSet` if one exists.

In the rest of this chapter we will address the problem of verifying that a given Java program satisfies the safety properties specified by an `Easl` specification.

6.3 Separation Strategies

The goal of a separation strategy is to separate or decompose a verification problem into a collection of verification subproblems. We now present an informal description of separation strategies. A more formal meaning will be given to separation strategies in Section 6.4.2.

Consider a typestate property, such as "an `InputStream` should not be `read` after it is `closed`". In this case, verification of the safety property for one `InputStream` object does not depend on the state of another `InputStream` object. Hence, the verification can be done independently for each `InputStream` object. This amounts to a very simple separation strategy.

Some safety properties, such as the `JDBC ResultSet` property, involve multiple related objects – we refer to these as *first-order safety properties*. Consequently, verification of such properties can be separated into subproblems in several different ways, each with potentially different efficiency and precision tradeoffs. Before we present some of the possible separation strategies, we introduce a simple

language for specifying a separation strategy.

In our approach, a separation strategy represents a method for *choosing* a set of objects. A set of chosen objects identifies a subproblem where verification is restricted to the chosen objects. For effective verification, a strategy should identify other objects that may have an impact on a chosen object and choose them too. This motivates the definition of the following language for specifying strategies.

An (atomic) separation strategy is a sequence of *choice* operations, where each choice operation identifies one or more objects that are chosen, as a function of previously chosen objects.

```
<atomic-strategy> ::= <choice-spec> *
<choice-spec> ::=
  choose (some|all) <var>:<constr> [/<condition>]
<constr> = <type-name> ( <var-list> )
```

Each choice operation consists of a variable name, a signature of a constructor, and an optional condition. The choice operation `choose some` performs a non-deterministic selection of objects, created through the specified constructor, that satisfy the condition. The operation `choose all` chooses all objects created through the specified constructor that satisfy the condition. Both choice operations evaluate the condition, and apply their choice on entry to the specified constructor. For simplicity, we assume that each type has a single constructor.

We now present some strategies for the `JDBC ResultSet` property.

Single Choice The motivation for our first strategy is the observation that there is no interaction between different `Connection`s: it should be possible to perform verification for each `Connection` independently. Hence, the following strategy performs separation at the level of a `Connection`.

```
choose some c :  Connection()
choose all s :  Statement(x) / x == c
choose all r :  ResultSet(y) / y == s
```

The separation strategy described above first non-deterministically chooses a single `Connection`, then proceeds by choosing *all* `Statement`s created from this `Connection`, and then choosing *all* `ResultSet`s created from these `Statement`s. For the running example, this amounts to separating the verification problem into two independent subproblems, one for each `Connection`.

Multiple Choice However, it should be clear from the JDBC specification that it is possible to perform a more fine-grained separation than the single choice strategy described above. In particular, the correct usage of a `ResultSet` does not really depend on how *any* other `ResultSet` is used. Thus, it is not necessary to perform verification of the different `ResultSet`s created, for isntance, from a single

`Statement` together. However, the correct usage of a `ResultSet` does depend on the `Statement` and `Connection` underlying the `ResultSet`. These observations motivate the following separation strategy.

```
choose some c :  Connection()
choose some s :  Statement(x) / x == c
choose some r :  ResultSet(y) / y == s
```

For the running example, this strategy produces a set of 5 subproblems, one for each combination of matching `Connection`, `Statement` and `ResultSet`.

Note that using a finer-grained separation strategy may or may not lead to more efficient verification. On one hand, finer-grained separation leads to smaller subproblems that can be verified more easily. On the other hand, it also leads to a larger number of subproblems. The relative performance of a strategy may depend on the amount of work that is duplicated across the different subproblems. The strategy we present next is likely to reduce the amount of work duplicated across subproblems.

Incremental The two strategies we have seen are examples of *atomic* strategies. In this chapter, we also explore the possibility of applying a sequence of increasingly complex separation strategies to perform verification. The motivation for this is simple: usually many verification subproblems may be amenable to simple and efficient verification, but some verification subproblems may require more precise analysis for successful verification.

An incremental strategy is a sequence of atomic strategies, which are tried one after another, stopping when one of the atomic strategies completely verifies the program. An atomic strategy can make use of failure information from the previous atomic strategy applied to the program. We restrict ourselves to a very simple form of failure information, where the choice operation can restrict attention to individuals that failed verification in the previous step. We will illustrate this with examples first, and later explain how these strategy specifications are interpreted.

```
{
  choose some r :  ResultSet(y)
} on failure {
  choose some s :  Statement(x)
  choose some failing r :  ResultSet(y) / y == s
} on failure {
  choose some c :  Connection()
  choose some failing s :  Statement(x) / x == c
  choose some failing r :  ResultSet(y) / y == s
}
```

The above strategy optimistically first attempts to verify usage of each `ResultSet` independent of even the `Statement` underlying the `ResultSet`. If that fails, it then attempts to verify usage of `ResultSet`s, while tracking usage of the underlying `Statement`. If that too fails, it then attempts verification using even more context.

Note that an incremental strategy may be thought of as a very simple (fixed) iterative refinement scheme. For our running example, the very first atomic strategy in the sequence above successfully verifies all correct uses of `ResultSet`.

Semantics and Correctness Note that the language presented above is powerful enough to specify *partial* verification problems, where the checking is done only for the specified *subset* of objects. This power is useful in some contexts. However, the goal of a *strategy* is typically to improve the precision and efficiency of verification but not affect its correctness. In order for a separation strategy to guarantee correctness, it has to *cover* all objects of the types being verified.

We later describe how a strategy specification defines an instrumented semantics for a program: every program-state in the standard semantics corresponds to a set of instrumented-program-states in the instrumented semantics, where an instrumented-program-state may be roughly thought of as a program-state plus a set of objects in the program-state (which are the "chosen" objects). A strategy is said to completely cover a type T if for every program-state σ in the standard semantics, and for every object obj of type T in σ, there exists an instrumented-program-state in which obj is a chosen object.

Theorem 6.3.1 *A separation strategy that consists only of choice operations with no condition and choice operations of the form:*

`choose all x : T (`$\mathtt{w}_1$`,...,`$\mathtt{w}_k$`) / (`$\mathtt{w}_i$ `== `$\mathtt{z}_j$`)`

where $\mathtt{w}_i$ *(*$1 \leq i \leq k$*) is a parameter of the constructor* `T`*, and* $\mathtt{z}_j$ *is a variable bound by earlier choice operations, completely covers* `T`*.*

6.4 Separation

In this section, we show how a separation strategy is utilized to decompose a verification problem into a set of verification subproblems. We first illustrate how an `Easl` safety property specification and a Java program together can be translated into an analysis problem instance in the parametric analysis framework of [91]. We then show how an `Easl` safety property specification, a Java program, and a separation strategy specification together can be translated into a *modified* analysis problem instance (corresponding to a set of verification subproblems). (This translation provides the semantics of a separation strategy.)

Predicates	**Intended Meaning**
$eq(v_1, v_2)$	v_1 equals to v_2
$x(v)$	reference variable x points to the object v
$fld(v_1, v_2)$	field fld of the object v_1 points to the object v_2
$bv()$	boolean variable bv has true value
$bf(v)$	boolean field bf holds for object v
$site[AS](v)$	object v was allocated in allocation site AS

Table 6.1: Predicates for partial Java semantics.

6.4.1 Background

We now present an overview of *first-order transition systems* (FOTS), the formalism underlying the parametric analysis framework of [91]. FOTS may be thought of as an imperative language built around an expression sub-language based on first-order logic

In a FOTS, the state of a program is represented using a first-order logical structure in which each individual corresponds to a heap-allocated object and predicates of the structure correspond to properties of heap-allocated objects.

Definition 6.4.1 *A 2-valued logical structure over a set of predicates P is a pair $C^\natural = \langle U^\natural, \iota^\natural \rangle$ where:*

- *$U^\natural$ is the universe of the 2-valued structure. Each individual in $U^\natural$ represents a heap-allocated object.*
- *$\iota^\natural$ is the interpretation function mapping predicates to their truth-value in the structure: for every predicate $p \in P$ of arity k, $\iota^\natural(p) : {U^\natural}^k \rightarrow \{ 0, 1 \}$.*

In the following we will use $p(v)$ as shorthand for $\iota^\natural(p)(v)$ when no confusion is likely.

Table 6.1 shows some of the predicates we use to record properties of individuals in this chapter. A unary predicate $x(v)$ holds when the reference (or pointer) variable `x` points to the object v. Similarly, a binary predicate $fld(v_1, v_2)$ records the value of a reference (or pointer-valued) field `fld`. A nullary predicate $bv()$ records the value of a local boolean variable `bv` and a unary predicate $bf(v)$ records the value of a boolean field `bf`. Finally, a unary predicate $site[AS](v)$ records the allocation site AS in which an object was allocated.

In order to enable interprocedural analysis we explicitly represent stack frames and a corresponding set of predicates following [89]. Since this does not interfere with the material in this chapter, to simplify presentation we do not describe these predicates.

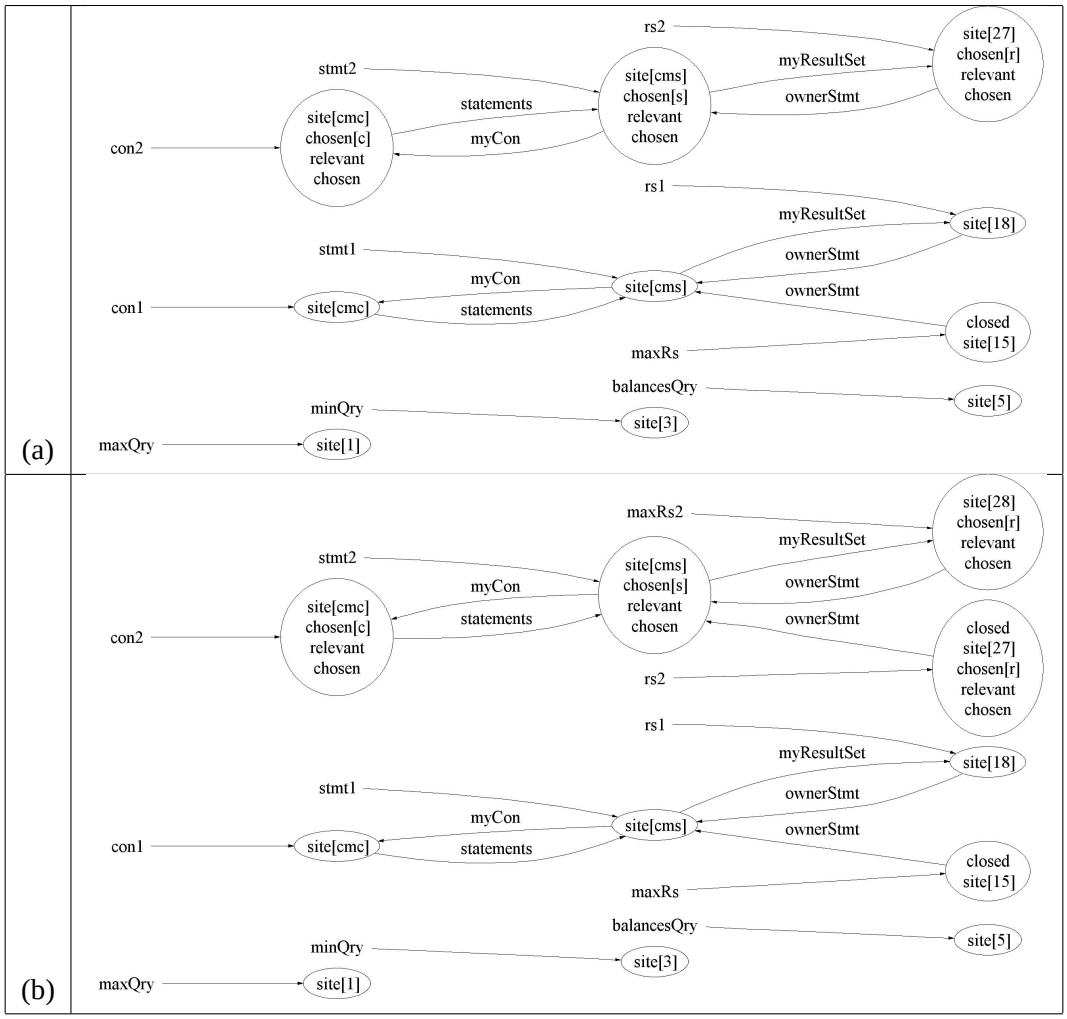


Figure 6.5: Concrete program configurations representing a possible program state (a) at line `28` and (b) after execution of the statement at line `28`

In this chapter, program configurations are depicted as directed graphs. Each individual of the universe is drawn as a node. A unary predicate $p(o)$ which holds for a node u is drawn inside the node u. A binary predicate $p(u_1, u_2)$ which evaluates to 1 is drawn as a directed edge from u_1 to u_2 labeled with the predicate symbol.

Example 6.4.2 *Fig. 6.5 shows a concrete program configuration representing a global state of the program before executing the statement at line `28`. In this configuration, three `String` objects were allocated in the heap and are referenced by `maxQry`, `minQry`, `balancesQry`. The configuration also contains two `Connection` objects referenced by `con1` and `con2`, two `Statement` objects referenced by `stmt1` and `stmt2`, and three `ResultSet` objects referenced by `maxRs`, `rs1`, and `rs2`. Note that the `ResultSet` referenced by `maxRs` is closed. The meaning of the predicates $relevant(u)$, $chosen[c](u)$, $chosen[s](u)$, and $chosen[r](u)$ will become clear in the next section.*

Predicates	**Intended Meaning**
$chosen[x](v)$	object v was chosen by choice operation for strategy variable x
$wasChosen[x]()$	some object was chosen for strategy variable x
$chosen(v)$	object v was chosen by some choice operation
$relevant(v)$	abstraction-directing predicate recording relevant objects

Table 6.2: Additional predicates of the instrumented semantics.

6.4.2 Instrumentation For Separation

In this section we explain how we translate a Java program, an `Easl` specification, and a strategy specification into a FOTS. Specifically, the strategy specification is used to instrument the standard translation of a Java program and `Easl` specification into a FOTS. (This translation also directly provides a formal semantics for a separation strategy as a method for non-deterministically choosing a set of objects during program execution.) We use the predicates in Table 6.2 to instrument the semantics. Predicates of the form $chosen[x](v)$, $wasChosen[x]()$, and $chosen(v)$ are used to express the separation strategy. The predicate $relevant(v)$ is an abstraction-directing predicate that controls the way in which an object is abstracted.

Consider a choice operation

```
choose all x :  T (w1,...,wi) / e(w1,...,wi,z1,...,zk)
```

Here, we say that the choice operation binds variable `x`. Variables $\texttt{w}_1$ through $\texttt{w}_i$ are free variables corresponding to parameters of a call to a constructor for type `t`, while $\texttt{z}_1$ through $\texttt{z}_k$ are variables bound by earlier choice operations. In order to model the specified choice operation, we introduce an instrumentation predicate $chosen[\texttt{x}](u)$. The idea is for the predicate $chosen[\texttt{x}](u)$ to hold true for exactly the objects that are chosen by the above choice operation. We achieve this by translating the condition `e(...)` specified for the choice operation into a first-order logic formula which is evaluated on entry to the specified constructor of `t` to compute the value of $chosen[\texttt{x}](u)$ for the newly created object u. (Technically, this translation works by converting the free occurrences of a variable $\texttt{z}_j$ by occurrences of an existentially quantified logical variable O_j that is constrained to satisfy predicate $chosen[\texttt{z}_j](O_j)$.)

The translation of a `choose some x` operation is similar, except that the translation ensures that at most one of the objects that is eligible for selection by the operation is chosen. This is done by introducing a second instrumentation predicate $wasChosen[\texttt{x}]()$ that indicates if an object has already been selected during program execution for the corresponding choice operation (thus, it is defined by the instrumentation formula $\exists O.chosen[\texttt{x}](O)$). When a new `t` object O is constructed, $chosen[\texttt{x}](O)$ is

set to false if $wasChosen[\mathtt{x}]()$ evaluates to true or if the selection formula corresponding to the choice operation evaluates to false. Otherwise, $chosen[\mathtt{x}](O)$ is non-deterministically assigned either true or false, and $wasChosen[\mathtt{x}]()$ is correspondingly updated.

Given a simple strategy specification consisting of n choice operations over variables $\mathtt{z}_1$ through $\mathtt{z}_n$, we also introduce a unary predicate $chosen(O)$ that indicates if an object was chosen by any of the n choice operations: thus, it is defined by the instrumentation formula $chosen[\mathtt{z}_1](O) \vee \cdots \vee chosen[\mathtt{z}_n](O)$.

Finally, the actual checks on objects that verify they satisfy the necessary preconditions when methods are invoked on them, are instrumented to perform the checks only for chosen objects.

For now, the predicate $relevant(u)$ may be thought of as being equivalent to $chosen(u)$. We will later see that the set of relevant objects includes all the chosen objects and potentially some other objects as well.

Example 6.4.3 *The single-choice strategy for JDBC is modelled using predicates $chosen[c](u)$, $chosen[s](u)$, and $chosen[r](u)$. Upon entry to the constructor `Statement(Connection c)`, the condition of the corresponding choice operation is evaluated and the `Statement` is chosen if the passed `Connection` is the one for which $chosen[c](u)$ holds. Similarly, the condition for choosing a `ResultSet` is evaluated on entry to constructor `ResultSet(Statement s)`. As a result, for each subproblem $chosen[c](u)$ holds for (at most) a single `Connection` component, and $chosen[s](u)$, $chosen[r](u)$ hold for `Statement`s and `ResultSet`s that are related to the chosen `Connection`. Part of the instrumented program for this strategy is shown in Fig. 6.6 (For clarity, we use `Easl` syntax to present the instrumented program).*

We now briefly indicate how incremental strategies are handled. The notion of a failed individual is fairly straightforward. A single strategy specification produces multiple verification subproblems, each over a set of chosen individuals. An individual is said to be a failed individual if it is a chosen individual of a verification subproblem that fails verification. However, we want to utilize simple strategy specifications that restrict their attention to individuals that failed the previous simple strategy specification. In general, this requires instrumentation that can identify at object-allocation time whether the allocated object corresponds to a failed individual in the previous verification step. This is hard to do in a very general way, and we restrict ourselves to allocation-site based identification of failed individuals: thus, if any one individual allocated at an allocation site fails verification, then all individuals allocated at that site are treated as failed individuals in the next verification step.

Operational Semantics

In a FOTS, program statements are modeled by *actions* that specify how the statement transforms an incoming logical structure into an outgoing logical structure. This is done primarily by defining the

```
class Connection {
 ...
 Connection() {
  if (!wasChosen) {
   if (?)  {
    chosen = true;
    wasChosen = true;
   } else
    chosen = false;
  }
  closed = false;
  statements = {};
 }
 Statement createStatement() {
  if (chosen)
   requires !closed;
  Statement st = new Statement(this);
  statements = statements U { st };
  return st;
 }
 ...
}
```

```
class Statement {
 ...
 Statement(Connection c) {
  chosen = c.chosen
  closed = false;
  myConnection = c;
  myResultSet = null;
 }
 ResultSet executeQuery(String qry) {
  if (chosen)
   requires !closed;
  if (myResultSet != null)
   myResultSet.closed = true;
  myResultSet = new ResultSet(this);
  return myResultSet;
 }
 ...
}
```

```
class ResultSet {
 ...
 ResultSet(Statement s) {
  chosen = s.chosen;
  closed = false ;
  ownerStmt = s;
 }
 ...
 boolean next() {
  if (chosen)
   requires !closed;
 }
}
```

Figure 6.6: An instrumented `Easl` specification for a simplified subset of the JDBC API with single-choice separation strategy.

values of the predicates in the outgoing structure using first-order logic formulae with transitive closure over the incoming structure [91].

Example 6.4.4 *Fig. 6.5(b) shows the effect of the statement* `maxRs2 = stmt2.executeQuery(maxQry)` *at line* `28`*, where the statement is applied to the configuration in Fig. 6.5. The effect of the statement is reflected by its updates to predicate values. Here, we assume that the choice predicates and the instrumentation predicates are updated according to the single-choice strategy of Section 6.3. Since the constructor of the new* `ResultSet` *is invoked with a chosen* `Statement` *object, the choice condition is satisfied and the newly created* `ResultSet` *is chosen and made relevant.*

6.4.3 Additional Instrumentation

The predicate *relevant* is intended to identify objects that must be modeled precisely for a verification subproblem. The separation strategy specification allows users to identify relevant objects (via choice clauses). An analysis designer, or a component library designer, can create separation strategies that reflect the dependencies that exist among component library objects, while an end user can create separation strategies that provide more dependency information (specific to their own program).

Currently, however, we do not assume that such extra dependency information will be available from

an end user. Instead, we rely on a more automatic approach that considers objects which reach a relevant object as relevant themselves, thus creating a notion of *transitive relevance*. Transitive relevance causes all objects that are on a path to a relevant object to become relevant as well, thus separating heap paths that may reach a relevant object from heap paths that cannot.

We achieve this by defining the instrumentation predicate $relevant(u)$ to be true if and only if there is a path from u to some chosen object v (i.e., some object v for which $chosen(v)$ is true). We update this predicate using the techniques of [86].

6.5 Heterogeneous Abstraction

The essence of our separation-based verification is the following: first, a separation strategy is used to choose a set of objects (for a given program trace); second, we utilize specialized abstractions to perform verification for the chosen objects efficiently. These specialized abstractions represent the chosen objects much more precisely than the remaining objects. We refer to these abstractions as *heterogeneous* abstractions as they represent different parts of the heap with different degrees of precision. In this section we describe the abstractions we use for separation-based verification.

Abstract Program Configurations

The goal of an abstraction is to create a finite representation of a potentially unbounded set of 2-valued structures (representing heaps) of potentially unbounded size. The abstractions we use are based on 3-valued logic [91], which extends boolean logic by introducing a third value $1/2$ denoting values that may be 0 or 1.

Definition 6.5.1 *A 3-valued logical structure over a set of predicates P is a pair $C = \langle U, \iota \rangle$ where:*

- *U is the universe of the 3-valued structure. An individual in U may represent multiple heap-allocated objects.*
- *ι is the interpretation function mapping predicates to their truth-value in the structure: for every predicate $p \in P$ of arity k, $\iota(p) : U^k \rightarrow \{ 0, 1, 1/2 \}$.*

*An abstract configuration may include **summary nodes**, i.e., an individual which corresponds to one or more individuals in a concrete configuration represented by that abstract configuration. A summary node u has $eq(u, u) = 1/2$, indicating that it may represent more than a single individual.*

As in [91], the abstract interpretations we use work by abstracting the set of 2-valued structures that can arise at a program point by a set of 3-valued structures. However, this can be done in a number of ways as shown below.

Individual Merging The basic abstraction primitive used by [91] is that of *individual merging*: a larger structure s can be safely approximated by a smaller 3-valued structure by merging multiple individuals into one, and by approximating the predicate values appropriately. Given an equivalence relation $\equiv$ on individuals, let $s/\equiv$ denote the structure obtained by merging individuals of s that are $\equiv$-equivalent together.

The above primitive induces a function $\mathit{abs}_1[\equiv]$ that abstracts a set of 2-valued structures by a set of 3-valued structures, defined by $\mathit{abs}_1[\equiv](S) = \{s/\equiv \mid s \in S\}$. (Strictly speaking, $\mathit{abs}_1[\equiv](S)$ retains only a single representative of isomorphic structures, but we ignore the fine distinction between isomorphism and equality for the sake of simplicity.)

[91] utilizes the equivalence relation $\equiv_A$ induced by a set of unary predicates A (referred to as the *abstraction* predicates) defined as follows: $o_1 \equiv_A o_2$ iff $p(o_1) = p(o_2)$ for every $p \in A$.

Structure Merging Subsequently, TVLA [65] introduced more aggressive abstraction mechanisms based on the idea of *merging multiple structures* into one. Define the *union* $s_1 \cup s_2$ of two structures to be the structure whose universe is the disjoint union of the universes of s_1 and s_2, with the predicate interpretations of s_1 and s_2 extended appropriately. The union of a set of structures S is defined similarly. Structures are merged by first taking their union, and then merging individuals of the union along the lines indicated previously: define $\bigsqcup_{\equiv}(S)$ to be $(\bigcup S)/\equiv$.

Now, consider an equivalence relation $\simeq$ defined on *structures*, indicating which structures must be merged together, and an equivalence relation $\equiv$ defined on *individuals*. We can now define a parameterized abstraction function $\mathit{abs}_2[\simeq, \equiv](S)$ that first applies *individual merging* to every structure s in S, and then merges together the resulting structures that are $\simeq$-equivalent. Formally, $\mathit{abs}_2[\simeq, \equiv](S)$ is defined to be:

$$\{ \bigsqcup\nolimits_{\equiv}(C) \mid C \text{ is an } \simeq\text{-equivalence class of } \mathit{abs}_1[\equiv](S) \}$$

TVLA utilizes the following $\simeq$ definitions: (a) $s_1 \simeq s_2$ iff s_1 and s_2 are isomorphic, (b) $s_1 \simeq s_2$ iff s_1 and s_2 have the same values for a specified set B of *nullary abstraction* predicates, (c) $s_1 \simeq s_2$ iff s_1 and s_2 have the same universes (modulo $\equiv$).

TVLA utilizes an extra unary predicate *active*, which indicates if an individual definitely exists in the universe or not, so that the structure $\bigsqcup_{\equiv}(S)$ can be used as an abstraction of every structure in S. Thus, if S is a set of 2-valued structures, then the predicate *active* is true for an individual o in $\bigsqcup_{\equiv}(S)$ iff the equivalence class represented by o includes at least one individual from every structure in S.

Heterogeneous Abstraction

Separation creates the possibility for achieving better efficiency by adapting the abstractions to model chosen individuals more precisely and the other individuals less precisely. In particular, this can be done by:

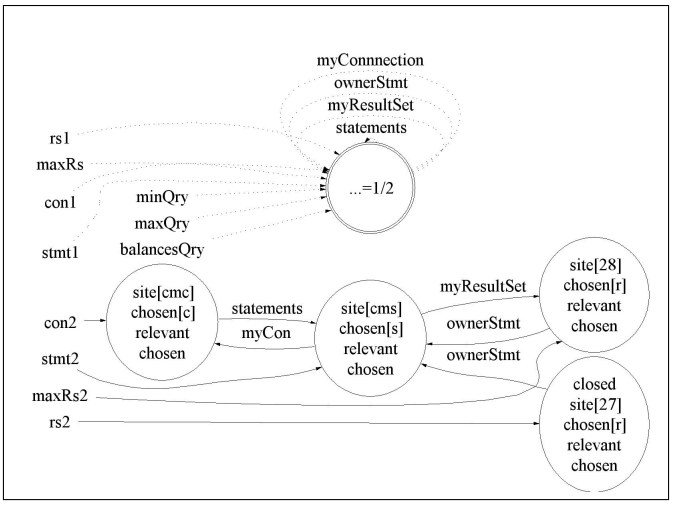


Figure 6.7: An abstract program configuration representing the concrete configuration of Fig. 6.5(b).

- *Adapting individual merging*: We can make finer distinctions between chosen individuals than between unchosen individuals, when we decide which individuals should be merged together. For instance, we can choose to use the less expensive allocation-site based merging for unchosen individuals, and more expensive variable-name based merging for chosen individuals.

- *Adapting structure merging*: Similarly, when deciding which structures should be merged into one, we could choose to treat chosen and unchosen individuals differently.

- *Adapting predicate values retained*: One could choose to not record the values of certain predicates for unchosen individuals. While this can reduce the space required to represent a structure, this does not, unlike the preceding techniques, reduce the number of structures in the abstraction. We will not discuss this issue in this chapter.

We now define a new family of equivalence relations for identifying individuals to be merged. Consider a quadruple $\langle c, A_1, A_0, A_{1/2}\rangle$ where c is a unary predicate, and A_1, A_0, and $A_{1/2}$ are all sets of unary predicates. The equivalence relation $\equiv_{\langle c,A_1,A_0,A_{1/2}\rangle}$ on individuals is defined by:

$$(c(o_1) = c(o_2) = 1) \wedge \forall p \in A_1.p(o_1) = p(o_2)) \vee$$
$$((c(o_1) = c(o_2) = 0) \wedge \forall p \in A_0.p(o_1) = p(o_2)) \vee$$
$$((c(o_1) = c(o_2) = 1/2) \wedge \forall p \in A_{1/2}.p(o_1) = p(o_2))$$

Given a set Γ of such tuples, we define $\equiv_\Gamma$ to be $\bigsqcap_{\gamma\in\Gamma} \equiv_\gamma$.

We similarly define a new criteria for structure merging. Given a unary predicate c, define $s_1 \simeq_c s_2$ iff the substructures of s_1 and s_2 consist only of individuals i for which $c(i) = 1$ are isomorphic.

For our separation-based verification, we utilize the abstraction induced by the equivalence relations $\equiv_{\langle relevant,A,\emptyset,A\rangle}$ and $\simeq_{relevant}$, where A is the set of abstraction predicates utilized by the underlying

separation-less verification. (In our implementation, this consists of the set of unary predicates).

Implementation Notes Our current implementation uses a very close approximation of the individual merging induced by the equivalence relation $\equiv_{\langle relevant,A,\emptyset,A\rangle}$ as follows: for every predicate p in A, we introduce a new instrumentation predicate $p_r(o) = p(o) \wedge relevant(o)$, and use the set of predicates $\{\ p_r \mid p \in A\ \}$ as the set of abstraction predicates.

Example 6.5.2 *Fig. 6.7 shows an abstract configuration representing the concrete configuration of Fig. 6.5(b), obtained by heterogeneous relevance-based abstraction. Abstract program configurations are depicted similarly to concrete configurations with an additional representation of summary nodes as nodes with double-line boundaries, and a $1/2$-valued binary predicate as a dashed edge. All individuals for which $relevant$ holds are abstracted by the values of the predicates in A_1. Other individuals, for which $relevant$ does not hold, are merged into a single summary node since $A_0 = \emptyset$. In particular, this abstract configuration abstracts away the current state of objects related to* `Connection con1`*, including the state of* `Statement stmt1`*. In the figure, we use $\ldots = 1/2$ instead of listing all predicates that have $1/2$ value for the summary node.*

If we had used a "homogeneous" abstraction, the non-relevant objects would have been abstracted using the same set of predicates as the relevant objects (A_1), thus keeping the objects related to the `Connection` *referenced by* `con1` *with the same precision and cost, as the ones related to* `Connection` *referenced by* `con2`*. The ability to treat these structurally-similar objects very differently during analysis is a key to obtaining good results with our method.*

Abstract Semantics

We will now briefly describe the abstract semantics ("transfer functions") we utilize for program statements.

A key idea underlying [91] is that the actions defining a standard operational semantics for a program statement (as a transformer of 2-valued structures) also define a corresponding abstract semantics for the statement (as a transformer of 3-valued structures). This abstract semantics is simply obtained by reinterpreting logical formulae using a 3-valued logic semantics and serves as the basis for an abstract interpretation. However, [91] also presents techniques, such as materialization, that improve the precision of such an abstract semantics. We directly utilize the implementation of these ideas available in TVLA.

We described earlier (see Section 6.4.2) how we utilize instrumentation predicates to identify relevant objects. We currently also utilize instrumentation predicates to achieve a heterogeneous abstraction. We use the techniques in [86] for automatically generating, from the instrumentation formula, an instrumented abstract semantics for statements to update the values of these instrumentation predicates.

6.6 Prototype Implementation

We have implemented a prototype of the separation verification engine using TVLA [65]. To translate Java programs and their specifications to TVP (TVLA input language) we have extended an existing Soot-based [105] front-end for Java developed by R. Manevich.

The implementation emulates heterogeneous abstraction using instrumentation predicates in TVLA, which adds some overhead. We believe that a native implementation of heterogeneous abstraction will yield better performance.

We applied our framework to verify various specifications for a number of example programs. Our specifications include correct usage of JDBC, IO streams, Java collections and iterators, and additional small but interesting specifications. The experiments were performed on a machine with a 1 Ghz Pentium 4 processor, and 1 GB RAM[2]. Results are shown in Table 6.3. The column titled "mode" shows the analysis mode for each line in the table. Verification with TVLA with no separation is referred to as *vanilla* mode. "Rep. Err." shows the number of reported errors, while "Act. Err." shows the number of actual errors. When counting errors, we count all errors reported at the same program location as a single error.

Our implementation allows control over which subproblems are verified simultaneously. This allows verification of subproblems related to one (or more) allocation-sites separately (as a separate execution of the analysis) from other subproblems, reducing the maximal memory footprint of the verification.

Our implementation supports the following execution modes:

- vanilla—verification with TVLA with no separation.
- single—*single choice* separation strategy, where each subproblem is verified separately (as a separate execution of the analysis).
- sim—*single choice* separation strategy, where all subproblem are verified simultaneously (in a single execution of the analysis).
- multi—*multiple choice* separation strategy, where each subproblem is verified separately.
- inc—*incremental* separation strategy, where each subproblem is verified separately.

The space measurement shown in Table 6.3 for separation modes (*single*, *multi*, *incremental*) is the maximal space required for analyzing a single set of subproblems. The time is the accumulated time for analyzing all subproblems. The table also shows measurements for simultaneous verification of all subproblems using single-mode (*sim* mode). For the JDBC example, the simultaneous single-choice mode is identical to the non-simultaneous mode.

ISPath is a simple correct program manipulating input streams. InputStream5 is a heapful example program that manipulates input-streams in holder objects at an arbitrary depth of the heap. For this program, the vanilla version produces a false-alarm that is avoided by the separation-based analysis.

[2] SQLExecutor analyzed on a machine with a 2.79Ghz processor.

Program	Description	Mode	Line No.	Space (MB)	Time (Sec)	Rep. Err.	Act. Err.
ISPath	inp. streams	vanilla	71	9.17	145.5	0	0
	/ IOStreams	single		2.51	17.4	0	
		sim		3.94	12.3	0	
Input	inp. streams	vanilla	64	16.35	439	1	0
Stream5	holders	single		17.65	240	0	
	/ IOStreams	sim		21.35	202	0	
Input	inp. streams	vanilla	64	13.72	343	1	1
Stream5b	holders err	single		19.71	279	1	
	/IOStreams	sim		22.74	243	1	
Input	inp. streams	vanilla	66	37.17	1344	1	0
Stream6	holders	single		13.91	69.4	1	
	/ IOStreams	sim		12.14	51.3	1	
JDBC	extended	vanilla	149	33.43	2500	1	1
Example	example	single		28.71	1090	1	
	/ JDBC	multi		16	7340	1	
		inc		12.5	3579	1	
JDBC	extended	vanilla	153	32.8	2500	0	0
Example	example	single		28.8	1090	0	
fixed	/ JDBC	multi		29.5	7500	0	
		inc		25.7	3339	0	
db	SpecJVM98	vanilla	644	89.25	10454	0	0
	db	single		90	2500	0	
	/ IOStreams	sim		91.17	1496	0	
Kernel	Collections	vanilla	82	42.23	8321	1	1
Bench.1	benchmark	single		13.15	657	1	
	/ CMP	sim		13.84	255	1	
		multi		14.45	4552	1	
		inc		14.45	960	1	
Kernel	Collections	vanilla	146	–	–	–	1
Bench.3	benchmark	single		107.8	12098	1	
	/ CMP	sim		128.7	7588	1	
		multi		119	69631	1	
		inc		106	12881	1	
SQL	JDBC	vanilla	1297	–	–	–	0
Executor	framework	single		80.59	5028	0	
	/ JDBC	multi		72.64	4919	0	
		inc		42.68	412	0	

Table 6.3: Analysis results and cost for the benchmark programs.

This stems from the use of *transitive relevance*, which makes the separation-based analysis more precise (for the relevant objects). Generally, since the separation-based analysis is more focused, it may allow use of a more precise abstraction than the one that could be used when applied uniformly. InputStream5b is an erroneous version of InputStream5 containing a single error. InputStream6 is another variation of InputStream5.

JDBCExample is an extended version of the running example that uses 5 `Connection`s. The high running-time result for incremental mode in this case is affected by the fact that there is a small number of `Statement`s (1) and `ResultSet`s (up to 3) associated with each `Connection`. db is a program from SpecJVM98 performing multiple database functions on a memory resident database.

KernelBenchmark1 and KernelBenchmark3 are part of a benchmark suite for testing Collections and Iterators used in [84]. SQLExecutor is an open source JDBC framework. For this benchmark, vanilla verification failed to terminate after more than 5 hours, but incremental-mode successfully verified the program in 412 seconds. This is a result of the correct and relatively simple usage of JDBC objects in this benchmark.

In some benchmarks separation gained an overall performance increase, while in others the total verification time in some modes was larger than the time for vanilla-mode verification. In all cases, however, the average time for verifying a single subproblem was significantly lower than the time required for vanilla verification. Thus, separation may be useful for answering on-demand queries when one is only interested in checking whether an object (or a set of correlated objects) can produce an error. For example, while the total time for multi-mode and incremental-mode in the JDBC example was larger than the time required for vanilla-mode, the average time for verifying each subproblem was approximately 670 seconds.

One interesting future direction is to exploit separation for increasing performance by parallelizing verification of subproblems.

6.7 Extensions and Future Work

We have experimented with two classes of iterative refinement schemes for approximating the set of relevant objects for a subproblem: the first iteratively identifies more “relevant program variables” and turns objects pointed-to by these variables relevant; the second iteratively identifies “relevant allocation sites” and turns objects allocated at these sites relevant. Both classes of our refinement schemes are guaranteed to terminate (with all objects being relevant in the worst case), but are not guaranteed to yield a successful verification. Our initial experience indicates that these techniques work well for relatively small examples.

Chapter 7

Applications

In this chapter, we show several applications of our techniques for verifying non-trivial Java programs. Section 7.1 shows how to use our techniques for establishing local temporal heap properties, and use these for compile-time memory management. In Section 7.2, we apply our techniques to verify concurrent queue algorithms, which are in part implemented in the `java.util.concurrent` package of JDK1.5. We conclude this chapter with Section 7.3, describing our solution to the apprentice challenge, a Java verification challenge posed by J. Moore.

The theories which I have expressed there, and which appear to you to be so chimerical, are really extremely practical — so practical that I depend upon them for my bread and cheese.
–Sir Arthur Conan Doyle, *A Study in Scarlet*.

7.1 Compile-Time Memory Management

In this section, we present a framework for statically reasoning about temporal heap safety properties. We focus on *local temporal heap safety properties*, in which the verification process may be performed for a program object independently of other program objects (this kind of properties was referred to as *spatially separable* in Section 4.3). We apply our framework to produce new conservative static algorithms for compile-time memory management, which prove for certain program points that a memory object or a heap reference will not be needed further. These algorithms can be used for reducing space consumption of Java programs. We have implemented a prototype of our framework, and used it to verify compile-time memory management properties for several small, but interesting example programs, including JavaCard programs.

Research in this section was conducted in collaboration with R. Shaham, as part of his PhD thesis. A preliminary version of this research also appeared in [95]. In this section, we only describe parts of the research that are relevant to this thesis. In particular, we omit discussion of empirical results that could be found at [95].

7.1.1 Introduction

This work is motivated by the need to reduce space consumption, for example, for memory-constrained applications in a JavaCard environment. Static analysis can be used to reduce space consumption by identifying source locations at which a heap-allocated object is no longer needed by the program. Once such source locations are identified, the program may be transformed to directly free unneeded objects, or aid a runtime garbage collector collect unneeded objects earlier during the run.

The problem of statically identifying source locations at which a heap-allocated object is no longer needed can be formulated as a local temporal heap safety property — a temporal safety property specified for each heap-allocated object independently of other objects (this kind of properties was referred to as *spatially separable* in Section 4.3).

The contributions described in this section can be summarized as follows:

- We present a framework for verifying local temporal heap safety properties of Java programs.
- Using this framework, we formulate two important compile-time memory management properties that identify when a heap-allocated object or heap reference is no longer needed, allowing space savings in Java programs.
- We have implemented a prototype of our framework, and used it as a proof of concept to verify compile-time memory management properties for several small but interesting example programs, including JavaCard programs.
- We show that our heap abstraction is precise enough to verify interesting compile-time memory management properties, while other points-to based heap abstractions fail to verify our properties of interest.

Local Temporal Heap Safety Properties

This section describes a framework for automatically verifying *local temporal heap safety properties*, i.e., temporal safety properties that could be specified for a program object independently of other program objects. In this section, we refer to properties as being *local temporal heap safety* properties instead of *typestate* properties (as used in Chapter 3) to emphasize that the verification algorithms in this section handle typestate verification for programs with arbitrary aliasing relationships. The class of properties handled in this section is contained in the class of *spatially separable* properties (used in Section 4.3) since in this section we only address spatially separable safety properties.

We assume that a safety property is specified using a *heap safety automaton* (HSA), which is a deterministic finite state automaton. The HSA defines the valid sequences of events that could occur for a single program object.

During the analysis, events are triggered for state machines associated with objects. It is important to note that our framework implicitly allows infinite state machines, since the number of objects is unbounded, and a state machine is associated with every object. Thus, precise information on heap paths to disambiguate program objects is crucial for the precise association of an event and its corresponding program object's state machine.

Local temporal heap properties are properties that consider the temporal behavior of each object separately. This allows the verification algorithm to consider each object independently. In this section, this allows us to simplify the general rewrite rules of the ETL existential semantics (Definition 5.2.12) to a preconstructed automaton associated with each object. Furthermore, in this section we only consider safety properties, which allows us to simplify the acceptance criterion of the automaton using finite-automaton acceptance instead of a Büchi acceptance.

In this section, we develop static analysis algorithms that verify that on all execution paths, all objects are in an HSA accepting state. In particular, we show how the framework is used to verify properties that identify when a heap-allocated object or heap reference is no longer needed by the program. This information could be used by an optimizing compiler or communicated to the runtime garbage collector to reduce the space consumption of an application. Our techniques could also be used for languages like C to find a misplaced call to `free` that prematurely deallocates an object.

Compile-Time Memory Management Properties

Runtime garbage collection (GC) algorithms are implemented in Java and C# environments. However, GC does not (and in general *cannot*) collect all the garbage that a program produces. Typically, a GC collects objects that are no longer reachable from a set of *root* references. However, there are some objects that the program never accesses again and therefore not needed further, even though they are reachable. In [93, 94] Shaham et. al. show a potential of saving 39% of the space by freeing reachable unneeded objects. Moreover, in some applications, such as those for JavaCard, GC is avoided by employing static object pooling, which leads to non-modular, limited, and error-prone programs.

Existing compile-time techniques produce limited savings. For example, [2] produces a limited savings of a few percent due to the fact that its static algorithm ignores references from the heap. Indeed, our dynamic experiments indicate that the vast majority of savings require analyzing the heap.

In this section, we develop two new static algorithms for detecting and deallocating garbage objects:

free analysis Statically identify source locations at which it is safe to insert a free statement in order to deallocate a garbage element.

assign-null analysis Statically identify source locations at which it is safe to assign null to heap references that are not used further in the run.

The assign-null analysis leads to space savings by allowing the GC to collect more space. In [94] Shaham et. al. show that assigning null to heap references immediately after their last use has an average space-saving potential of 15% beyond existing GCs. Free analysis could be used with runtime GC in standard Java environments and without GC for JavaCard.

Both of these algorithms handle heap references and destructive updates. They employ both forward (history) and backward (future) information on the behavior of the program. This allows us to free more objects than reachability-based compile-time garbage collection mechanisms (e.g., [56]), which only consider the history.

A Motivating Example

Fig. 7.1 shows a program that creates a singly-linked list and then traverses it. We would like to verify that for this program a `free y` statement can be added immediately after line 10. This is possible because once a list element is traversed, it cannot be accessed along any execution path starting after line 10. It is interesting to note that even in this simple example, standard compile-time garbage collection techniques (e.g., [56]) will not issue such a free statement, since the element referenced by `y` is reachable via a heap path starting from `x`. Furthermore, integrating limited information on the future of the computation such as liveness of local reference variables (e.g., [2]) is insufficient for issuing such a free statement. Nevertheless, our analysis is able to verify that the list element referenced by `y` is no longer needed, by investigating all execution paths starting at line 10.

In order to prove that a free statement can be added after line 10, we have to verify that all program objects referenced by `y` at line 10 are no longer needed on execution paths starting at this line. More specifically, for every execution path and for every object o, we have to verify that from line 10 there is no use of a reference to o. In the sequel, we show how to formulate this property as a heap safety property and how our framework is used to successfully verify it.

A Framework for Verifying Heap Safety Properties

Our framework is conservative, i.e., if a heap safety property is verified, it is never violated on any execution path of the program. As usual for a conservative framework, we might fail to verify a safety property which holds on all execution paths of the program.

Assuming the safety property is described by an HSA, we instrument the program semantics to record the automaton state for every program object. First-order logical structures are used to represent a global state of the program. We augment this representation to incorporate information about the automaton state of every heap-allocated object.

Our abstract domain uses first-order 3-valued logical structures to represent an abstract global state of the program, which represents several (possibly an infinite number of) concrete logical structures [91].

```
  class L { // L is a singly linked list
    public L n; // next field
    public int val; // data field
  }
  class Main { // Creation and traversal of a singly-linked list
    public static void main(String args[]) {
      L x, y, t;
1     x = null;
2     while (...)  { // list creation
3       y = new L();
4       y.val = ...;
5       y.n = x;
6       x = y;
      }
7     y = x;
8     while (y != null) { // list traversal
9       System.out.print(y.val);
10      t = y.n;
11      y = t;
      }
    }
  }
```

Figure 7.1: A program for creating and traversing a singly linked list.

We use *canonical abstraction* that maps concrete program objects (i.e., individuals in a logical structure) to abstract program objects based on the properties associated with each program object. In particular, the abstraction is refined by the automaton state associated with every program object.

For the purpose of our analyses one needs to: (i) consider information on the history of the computation, to approximate the heap paths, and (ii) consider information on the future of the computation, to approximate the future use of references. Our approach here uses a forward analysis, where the automaton maintains the temporal information needed to reason about the future of the computation.

Outline

The rest of this section is organized as follows. In Section 7.1.2, we describe heap safety properties in general, and a compile-time memory management property of interest — the free property. Then, in Section 7.1.3, we give our instrumented concrete semantics which maintains an automaton state for every program object. Section 7.1.4 describes our property-guided abstraction and provides an abstract semantics. In Section 7.1.5, we describe an additional property of interest — the assign-null property, and discuss efficient verification of multiple properties.

7.1.2 Specifying Compile-Time Memory Management Properties via Heap Safety Properties

In this section, we introduce heap safety properties in general, and a specific heap safety property that allows us to identify source locations at which heap-allocated objects may be safely freed.

Informally, a heap safety property may be specified via a heap safety automaton (HSA), which is a deterministic finite state automaton that defines the valid sequences of events for a single object in the program. An HSA defines a prefix-closed language, i.e., every prefix of a valid sequence of events is also valid. This is formally defined by the following definition.

Definition 7.1.1 (Heap Safety Automaton (HSA)) *A heap safety automaton* $A = \langle \Sigma, Q, \delta, \text{init}, F \rangle$ *is a deterministic finite state automaton, where* Σ *is the automaton alphabet which consists of observable events,* Q *is the set of automaton states,* $\delta : Q \times \Sigma \rightarrow Q$ *is the deterministic transition function mapping a state and an event to a single successor state,* $\text{init} \in Q$ *is the* initial state, $\text{err} \in Q$ *is a distinguished* violation state *(the sink state), for which for all* $a \in \Sigma$, $\delta(err, a) = err$, *and* $F = Q \setminus \{\text{err}\}$ *is the set of accepting states.*

In our framework, an observable event is derived from the program state and the current statement. We assume the observable events are part of the specification. We associate an HSA state with every object in the program, and verify that on all program execution paths, all objects are in an accepting state. The HSA is used to define an instrumented semantics, which maintains the state of the automaton for each object. The automaton state is *independently* maintained for every program object. However, the same automaton is used for all program objects.

When an object o is allocated, it is assigned the initial automaton state. The state of an object o is then updated by automaton transitions corresponding to events associated with o, triggered by program statements. For example, an object o in automaton state q is updated by automaton transition α to have a new automaton state $\delta(q, \alpha)$, if o is associated with the observable event α occurring in the current program statement.

The states in the automaton capture history information on memory locations. Transitions in the automaton capture the changes in the history information when a statement corresponding to the event is executed. This can be formalized using trace semantics. To make the material more accessible, we use automata directly and define self-explanatory events.

Free Property

We now formulate the free property, which allows us to issue a free statement to reclaim objects unneeded further in the run. In the sequel, we make a simplifying assumption and focus on verification of

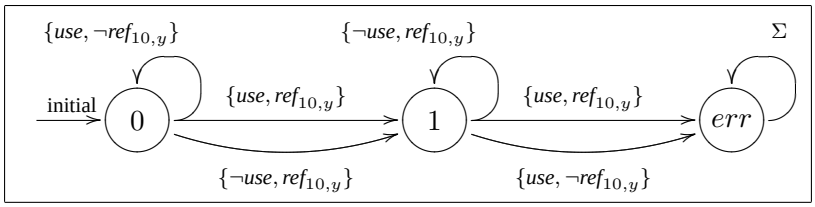


Figure 7.2: A heap safety automaton $A^{free}_{10,y}$ for free y at line 10.

the property for a single program point. In Section 7.1.5 we discuss a technique for efficient verification for a set of program points.

In order to formulate the free property we first consider the notions of a program state and a program trace. A *program state* $\sigma_i = \langle \text{store}_i, \text{pt}_i \rangle$ represents the global state of the program, which consists of the store (store_i) and the current program point (pt_i). A *trace* $\pi = \sigma_1, \sigma_2, \ldots$ is a (possibly infinite) sequence of program states σ_i. A trace reflects a program execution.

In order to define the *free* property, we also define the notion of *dynamic location liveness*.

Definition 7.1.2 (Dynamic Location Liveness) A memory location l **is dynamically live in a program state** σ_i **along a trace** π *if (i)* l *is used in* σ_j*, for some* $j \geq i$*, and (ii)* l *is not assigned in all* $\sigma_i, \ldots, \sigma_{j-1}$.

Intuitively, an object can be collected as soon as its references are no longer used. This observation leads to the following intuitive definition of the free property.

Definition 7.1.3 (Free Property $\langle pt, \text{x} \rangle$) The property free $\langle \text{pt}, \text{x} \rangle$ holds *if there exists no trace* π *with a program state* $\sigma_i = \langle store_i, pt \rangle$ *such that there exists a reference to the object referenced by* x *in* σ_{i+1}*, which is dynamically live in* σ_{i+1} *in* π.

The free property allows us to free an object that is not needed further in the run. In particular, when a free property $\langle pt, \text{x} \rangle$ holds for a program point pt and a reference variable x, it guarantees that it is *safe* to issue a `free(x)` statement immediately after pt. That is, it guarantees that adding such `free(x)` statement preserves the semantics of the original program (for a more formal treatment of semantic preserving transformations see [96]). Interestingly, such an object can still be reachable from a program variable through a heap path. For simplicity, we assume that a `free(x)` statement does nothing (and in particular does not abort) when x references the special `null` value.

Finally, for expository purposes, we only present the free property for an object referenced by a program variable. However, this free property can easily handle the free for an object referenced through an arbitrary reference expression `exp`, by introducing a new program variable `z`, assigned with `exp` just after pt, and verifying that `free(z)` may be issued just after the statement `z = exp`.

Free Property for the Running Example

Consider the example program of Fig. 7.1. We would like to verify that a `free y` statement can be added immediately after line 10, i.e., a list element can be freed as soon as it has been traversed in the loop.

The HSA $A^{free}_{10,y}$ shown in Fig. 7.2 represents the free property $\langle 10, y\rangle$. All states but err are accepting, and we therefore do not mark accepting states in the figure. The state labeled *err* is the automaton's violation state.

An arbitrary free property is formulated as a heap safety property using an HSA similar to the one shown in Fig. 7.2 where the program point and program variable are set accordingly. In particular, for a free property $\langle pt, \mathtt{x}\rangle$, the corresponding HSA $A^{free}_{pt,x}$ may be obtained from the automaton in Fig. 7.2 by replacing 10 with pt, and by replacing y with x.

The HSA could be automatically derived from an ETL specification such as Property 7.1. Formulating this property as a quantifier free formula (where free variables are interpreted as implicitly universally quantified), and using its negation with the rewrite rules of Definition 5.2.12 yields an automaton that is equivalent to the automaton of Fig. 7.2.

The alphabet of the automaton consists of sets of observable object attributes. For the purpose of verifying the free property, we maintain the following object attributes in the instrumented semantics (see Section 7.1.3) for an object o: (i) *use* attribute, which holds for o if the r-value of reference expression e (of the form `x` or of the form `x.f`) is used in the current statement execution, and the r-value of e is o, and (ii) $ref_{10,y}$ attribute, which holds for o if the program execution is immediately after execution of the statement at line 10 and `y` references o after the execution of the statement at line 10.

Based on the above object attributes we define the alphabet of the HSA $A^{free}_{10,y}$ to be

$$\Sigma = \{\{use, ref_{10,y}\}, \{use, \neg ref_{10,y}\}, \{\neg use, ref_{10,y}\}\}$$

For readability purposes, we show for a set of attributes (an alphabet symbol) the attributes that hold for an object as well as the attributes that do not hold for an object[1]. For example, the alphabet symbol $\{use, \neg ref_{10,y}\}$ denotes that the attribute *use* holds for an object (i.e., a reference to that object is used in the current statement), while the attribute $ref_{10,y}$ does not hold for that object (i.e., either the current statement is not at pt, or this object is not referenced by `y` after the current statement is executed). Finally, we use Σ in the self-loop emanating from the *err* state (see Fig. 7.2) as a shorthand expressing the fact that for all alphabet symbols the *err* state may only be transitioned to itself (i.e., when reaching the violation state, the automaton state cannot be changed, since the property is violated).

The HSA is in an accepting state along an execution path if and only if o can be freed in the program after line 10. Thus, when on all execution paths, for all program objects o, only accepting states are

[1] An equivalent way of writing the alphabet would be $\Sigma = \{\{use, ref_{10,y}\}, \{use\}, \{ref_{10,y}\}\}$, where only attributes that hold for an object are shown.

associated with o, we conclude that `free(y)` can be added immediately after line 10.

First, when an object is allocated, it is assigned the initial state of $A^{free}_{10,y}$ (state 0). Then, a use of a reference to an object o (the *use* attribute holds for o) when the program execution is not immediately after line 10 (the $ref_{10,y}$ attribute does not hold for o) does not change the state of $A^{free}_{10,y}$ for o (the self-loop on state 0 labeled with $\{use, \neg ref_{10,y}\}$ is taken). When the program is immediately after line 10 and y references an object o (the $ref_{10,y}$ attribute holds for o), o's automaton state is set to 1 (if the *use* attribute holds for o the labeled edge $\{use, ref_{10,y}\}$ is taken, otherwise if the *use* attribute does not hold for o then the labeled edge $\{\neg use, ref_{10,y}\}$ is taken). If a reference to o is used further, (i.e., in the subsequent program configurations along the execution path a reference to o is used), and o's automaton state is 1 the automaton state for o reaches the violation state of the automaton (either via the $\{use, ref_{10,y}\}$ edge or via the $\{use, \neg ref_{10,y}\}$ edge). In that case the property is violated, and it is not possible to add a `free y` statement immediately after line 10 since it will free an object that is needed later in the program. However, in the program of Fig. 7.1, references to objects referenced by `y` at line 10 are not used further, hence the property is not violated, and it is safe to add a `free y` statement at this program point. Indeed, in Section 7.1.4 we show how the *free* $\langle 10, y\rangle$ property is verified.

The above definition of the free property directly and naturally corresponds to the ETL property

$$\Box\, \forall v.at[pt] \wedge x(v) \rightarrow \bigcirc \Box\, \neg use(v) \tag{7.1}$$

In the formulation of this property, we use the combination of the predicate $at[pt]$ (that holds when program execution is at the program point pt) and the next temporal operator to achieve the same effect of using the $after[pt]$ predicate, as this exposes the temporal relationships in a manner closer to Definition 7.1.3.

7.1.3 Instrumented Concrete Semantics

We define an instrumented concrete semantics that maintains an automaton state for each heap-allocated object. As in previous chapters, we use first-order logical structures to represent a global state of the program. In this section, we augment this representation to incorporate information about the automaton state of every heap-allocated object. We then describe an operational semantics manipulating instrumented configurations.

We use the predicates of Table 7.1 to record information used by the properties discussed in this section. The nullary predicate *after*$[pt]()$ records the program location in a configuration and holds in configurations in which the program is immediately after line pt. The unary predicate $x(o)$ records the value of a reference variable `x` and holds for the individual referenced by `x`. The binary predicate $f(o_1, o_2)$ records the value of a field reference, and holds when the field `f` of o_1 points to the object o_2.

The predicates $use(o)$ and $ref_{pt,x}$ maintain the object attributes needed for triggering events in the HSA $A^{free}_{pt,x}$. We describe these object attributes more completely in Section 7.1.3 and Section 7.1.3

Predicates	**Intended Meaning**
after$[pt]()$	program execution is immediately after program point pt
$x(o)$	program variable x references the object o
$f(o_1, o_2)$	field f of the object o_1 points to the object o_2
use(o)	a reference to o is used in the current program statement
$\mathit{ref}_{pt,x}(o)$	o is referenced by x and the execution is immediately after pt
$s[q](o)$	the current state of o's automaton is q

Table 7.1: Predicates for partial Java semantics.

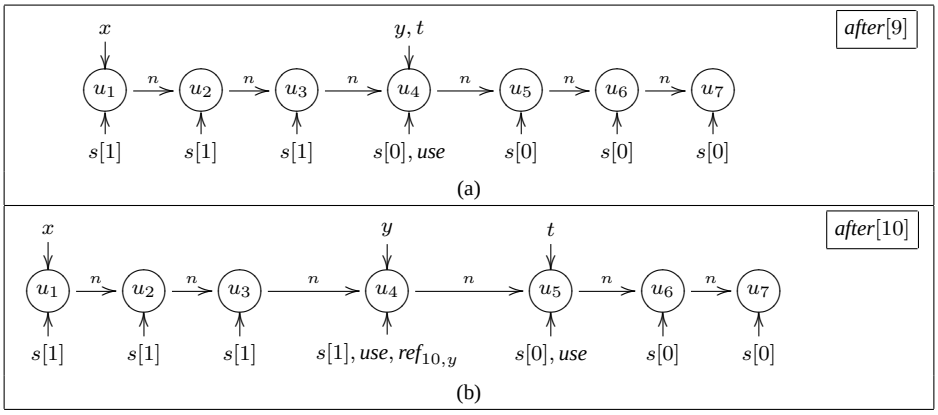


Figure 7.3: Concrete program configurations (a) before — and (b) immediately after execution of `t = y.n` at line 10.

Predicates of the form $s[q](o)$ (referred to as *automaton state predicates*) maintain temporal information by maintaining the automaton state for each object. Such predicates (corresponding to the *formula-predicates* of Section 5.4.1) record history information that is used to refine the abstraction. The abstraction is refined further by predicates that record spatial information, such as *reachability* and *sharing* (referred to as *instrumentation predicates* in [91]).

As in previous chapters, we depict program configurations as directed graphs. Each individual of the universe is displayed as a node. A unary predicate of the form $p(o)$ is shown as an edge from the predicate symbol to a node in which it holds. The name of a node is written inside the node using an *italic* face. Node names are only used for ease of presentation and do not affect the analysis. A binary predicate $p(u_1, u_2)$ which evaluates to 1 is drawn as directed edge from u_1 to u_2 labeled with the predicate symbol. Finally, a nullary predicate $p()$ is drawn inside a box.

Example 7.1.4 *The configuration shown in Fig. 7.3(a) corresponds to a global state of the program in which execution is immediately after line* 9*. In this configuration, a singly-linked list of* 7 *elements*

has been traversed up to the 4*-th element (labeled* u_4*) by the reference variable* y*, and the reference variable* t *still points to the same element as* y*. This is shown in the configuration by the fact that both predicates* $y(o)$ *and* $t(o)$ *hold for the individual* u_4*. Directed edges labeled by* n *correspond to values of the* n *field. The nullary predicate after*$[9]()$ *shown in a box in the upper-right corner of the figure records the fact that the program is immediately after line* 9*. The predicate use*(o) *holds for an object* o *if a reference to* o *is used in the current statement. For example, a reference to* u_4 *is used (due the use of* `y` *in the statement at line* 9*) thus we see an edge connecting use and* u_4*. The predicate ref*$_{10,y}$ *does not hold for any objects in this configuration, since the execution is not immediately after line* 10*. Finally, the predicates* $s[0](o)$ *and* $s[1](o)$ *record which objects are in state* 0 *of the automaton and which are in state* 1*. For example, the individual* u_3 *is in automaton state* 1 *and the individual* u_4 *is in automaton state* 0*.*

Operational Semantics

Program statements are modeled by generating the logical structure representing the program state after execution of the statement. First order logical formulae can be used to formally define the effect of every statement (see [91]). In particular, first-order logical formulae are used to model the change of the automaton state of every affected individual.

In general, the operational semantics associates a program statement with a set of HSA events that update the automaton state of program objects. The translation from the set of HSA events to first-order logical formulae reflecting the change of the automaton state of every affected individual is automatic. We now show how program statements are associated with $A^{free}_{pt,x}$ events. For expository purposes, and without loss of generality, we assume the program is normalized to a 3-address form. In particular, a program statement may manipulate reference expressions of the form `x` or `x.f`.

Object Allocation:

For a program statement `x = new C()`, a new object o_{new} is allocated, which is assigned the initial state of the HSA, i.e., we set the predicate $s[init](o_{new})$ to 1.

Example 7.1.5 *Consider the HSA* $A^{free}_{10,y}$ *of the example in Section 7.1.2. For this HSA we define a set of predicates* $\{s[0](o), s[1](o), s[err](o)\}$ *to record the state of the HSA individually for every heap-allocated object. Initially, when an object* o *is allocated at line* 3 *of the example program, we set* $s[0](o)$ *to 1, and other state predicates of* o *to 0.*

Maintaining the *use* attribute

The *use* attribute reflects information for an object depending on the current state of the program. Thus, conceptually, this means that before executing a statement the *use* attribute is set to *false* for all program

statement	**use-attribute is set to *true* for an object referenced by**
x = y	y
x = y.f	$y, y.f$
x.f = null	x
x.f = y	x, y
x binop y	x, y

Table 7.2: Use-attributes set by program statements.

objects, and then the *use* property is set to *true* for some of the objects depending on the executed program statement, as shown in Table 7.2.

In general, a use of `x` in a program statement updates the *use*(o) attribute to 1 for the object referenced by `x`. In addition, a use of the field `f` of the object referenced by `x` in a program statement updates *use*(o) attribute to 1 for object referenced by `x.f`. For example, as shown in Table 7.2, the statement `x = y.f` sets *use*(o) to 1 for the objects referenced by y and $y.f$.

Maintaining the $ref_{pt,x}$ attribute

As in the case of the *use* attribute, the $ref_{pt,x}$ attribute reflects information for an object depending on the current state of the program. Thus, conceptually, this means that before executing a statement the $ref_{pt,x}$ attribute is set to *false* for all program objects, and then this property is set to *true* for some of the objects depending on the executed program statement. In particular, we set the $ref_{pt,x}$ attribute to true for the object referenced by x when the execution is immediately after pt (i.e., when the currently executed statement is at program point pt). For example, for the $ref_{10,y}$ attribute, $ref_{10,y}(o)$ is set to 1 for the object referenced by y, when the execution is immediately after line 10.

Maintaining $s[q]$ predicates

We can now determine the transition taken in the automaton for an object o changing its associated automaton state from q_i to q_j. The idea is that an edge emanating from q_i is taken if the label on that edge matches the values of o's *use*, $ref_{pt,x}$ attributes. For example, in our running example, if an object o is associated with state 0, and both *use*, $ref_{10,y}$ attributes hold for o, then the edge labeled $\{use, ref_{10,y}\}$ connecting state 0 to state 1 (see Fig. 7.2) is taken, updating $s[0](o)$ to 0, and $s[1](o)$ to 1. In general, a transition from state q_i to state q_j for an object o is reflected by setting $s[q_i](o)$ to 0, and setting $s[q_j](o)$ to 1.

Example 7.1.6 *Fig. 7.3 shows the effect of the* `t = y.n` *statement at line* 10*, where the statement is*

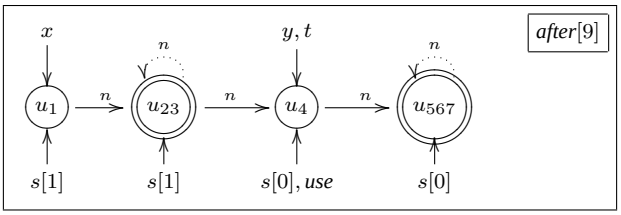


Figure 7.4: An abstract program configuration representing the concrete configuration of Fig. 7.3(a).

applied to the configuration labeled by (a). First, this statement updates the predicate $t(o)$ *to reflect the assignment by setting it to* 1 *for* u_5*, and setting it to* 0 *for* u_4*. In addition, it updates the program point by setting after*$[10]()$ *to* 1 *and after*$[9]()$ *to* 0*. Then, use*(o) *is set to* 1 *for both* u_4, u_5*. This is due to the use of* y *and* $y.f$ *in this statement. Also, ref*$_{10,y}(o)$ *is set to* 1 *for* u_4*, since the execution is after line* 10 *and* u_4 *is referenced by* y *at that time.*

We can now update the automaton states associated with program objects. For u_4 *the current associated automaton state is* 0*. The attributes use, ref*$_{10,y}$ *hold for* u_4*; thus, the* $\{$*use, ref*$_{10,y}\}$ *edge connecting automaton state* 0 *to automaton state* 1 *is taken, updating* $s[0](u_4)$ *to* 0*, and* $s[1](u_4)$ *to* 1*. In addition, for* u_5*, the attribute use holds, and the attribute ref*$_{10,y}$ *does not hold, thus the* $\{$*use*, $\neg$*ref*$_{10,y}\}$ *edge connecting state* 0 *to itself is taken, leaving* $s[0](u_5)$ *unchanged with the value* 1*.*

7.1.4 An Abstract Semantics

In this section, we present a conservative abstract semantics [25] abstracting the concrete semantics of Section 7.1.3.

As in earlier sections, we conservatively represent multiple concrete program configurations using a single logical structure with an extra truth-value $1/2$ that denotes values that could be 1 or could be 0.

Example 7.1.7 *The abstract configuration shown in Fig. 7.4 represents the concrete configuration of Fig. 7.3(a). The summary node labelled by* u_{23} *represents the linked-list items* u_2 *and* u_3*, both having the same values for their unary predicates. Similarly, the summary node* u_{567} *represents the nodes* u_5*,* u_6*, and* u_7*.*

Note that this abstract configuration represents many configurations. For example, it represents any configuration in which program execution is immediately after line 10 *and a linked-list with at least* 4 *items has been traversed up to some item after the third item.*

Note that since automaton states are represented using unary predicates, the abstraction is refined by the automaton state of each object. This provides a simple property-guided abstraction since individuals at different automaton states are not summarized together. Indeed, adding unary predicates to the abstraction increases the worst-case cost of the analysis. However, as noted in [91] in practice

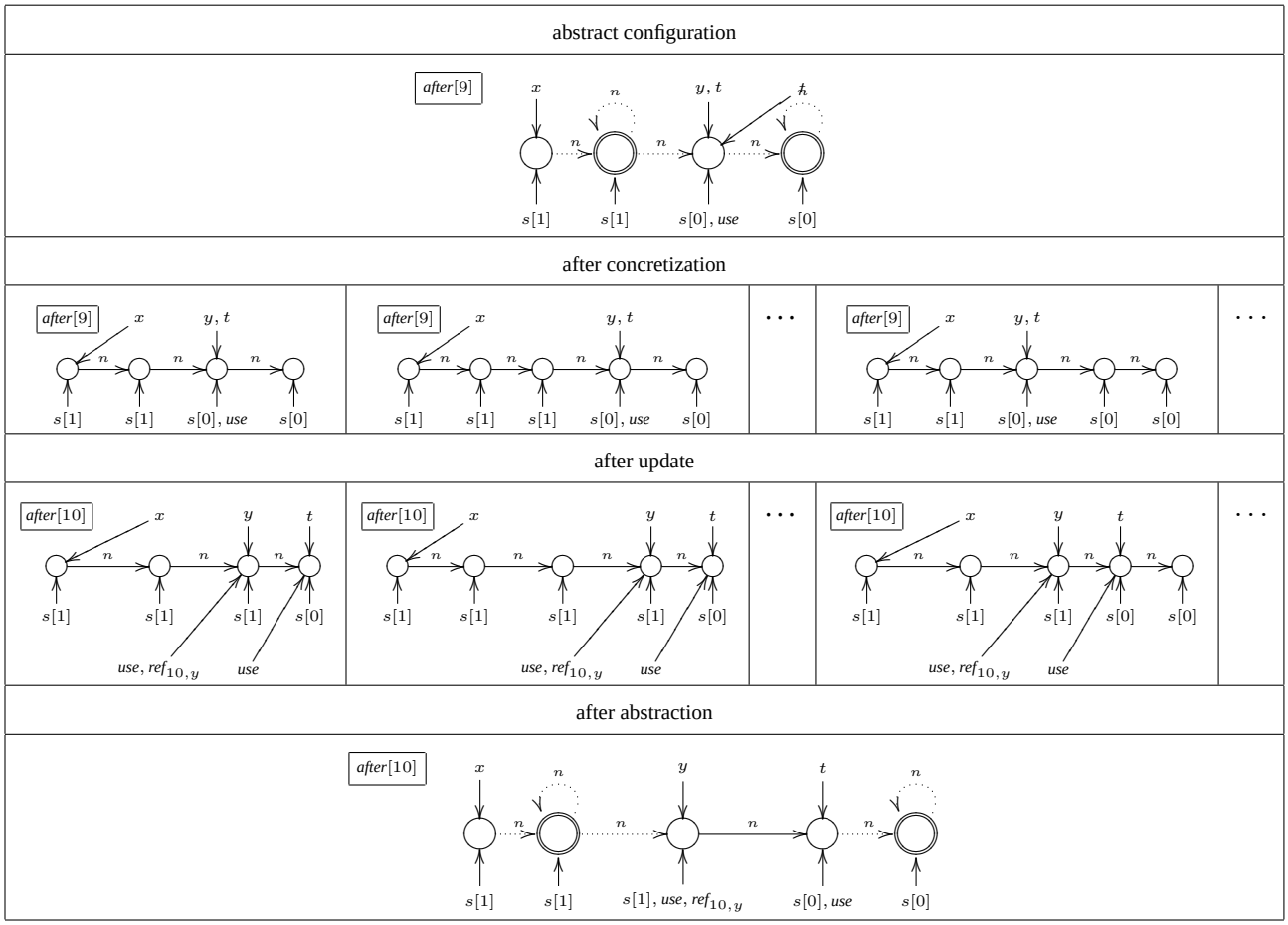


Figure 7.5: Concretization, predicate-update including automaton transition updates, and abstraction for the statement `t = y.n` at li ne 10.

this abstraction refinement often decreases significantly the cost of the analysis. Finally, our analysis is *relational*, allowing multiple 3-valued logical structures at a single program point, reflecting different behaviors.

Implementing an abstract semantics directly manipulating abstract configurations is non-trivial since one has to consider all possible relations on the (possibly infinite) set of represented concrete configurations. The following example conceptually shows how an action is applied directly to abstract configurations.

Example 7.1.8 *Fig. 7.5 shows the stages of an abstract action: first, concretization is applied to the abstract configuration resulting in an infinite set of concrete configuration represented by it. The program statement update is then applied to each of these concrete configurations. The program statement update also includes the update of the use and* $ref_{pt,x}$*, attributes, and the application of automaton transition updates described in Section 7.1.3. That is, the* use *attribute is set to* 1 *for the objects referenced*

by `y` *and* `y.n`*, and the* $ref_{10,y}$ *attribute set to* 1 *for the object referenced by* `y`*. Then,* $s[1]$ *is set to* 1 *for the object referenced by* `y`*, and* $s[0]$ *is set to* 0 *for the object referenced by* `y`*. Finally, after all transition updates have been applied, the resulting concrete configurations are abstracted resulting in a finite representation.*

Our prototype implementation described in [95] operates directly on abstract configurations using *abstract transformers*. The implemented actions are more conservative than the ones obtained by the best transformers. Interestingly, since temporal information is encoded as part of the concrete configuration via automaton state predicates, the soundness of the abstract transformers is still guaranteed by the *Embedding Theorem* of [91]. Our experience shows that the abstract transformers used in the implementation are still precise enough to allow verification of our heap safety properties.

When the analysis terminates, we verify that in all abstract configurations, all individuals are associated with an accepting automaton state, i.e., in all abstract configurations, for every individual o, the predicate $s[err](o)$ evaluates to 0. The soundness of our abstraction guarantees that this implies that in all concrete configurations, all individuals are associated with an accepting automaton state, and we conclude that the property holds.

7.1.5 Extensions

In this section, we extend the applicability of our framework by: (i) formulating an additional compile-time memory management property — the assign-null property; and (ii) extending the framework to simultaneously verify multiple properties.

Assign-Null Analysis

The assign-null problem determines source locations at which statements assigning null to heap references can be safely added. Such null assignments lead to objects being unreachable earlier in the program, and thus may help a runtime garbage collector collect objects earlier, thus saving space. As in Section 7.1.2, we show how to verify the assign-null property for a single program point and discuss efficient verification for a set of program points in the next section.

Definition 7.1.9 (Assign-Null Property $\langle pt, \mathtt{x}, \mathtt{f}\rangle$) The property assign-null $\langle pt, \mathtt{x}, \mathtt{f}\rangle$ holds *if there exists no trace* π *that includes a program state* $\sigma_i = \langle store_i, pt\rangle$ *such that the location denoted by* $x.f$ *in* σ_{i+1} *is dynamically live in* σ_{i+1} *in* π*.*

The assign-null property allows us to assign null to a dead heap reference. In particular, when an assign-null property $\langle pt, \mathtt{x}, \mathtt{f}\rangle$ holds for a program point pt, a reference variable `x` and a reference field `f`, it guarantees that it is *safe* to issue a `x.f = null` statement immediately after pt. That is, it guarantees that adding such `x.f = null` statement preserves the semantics of the original program

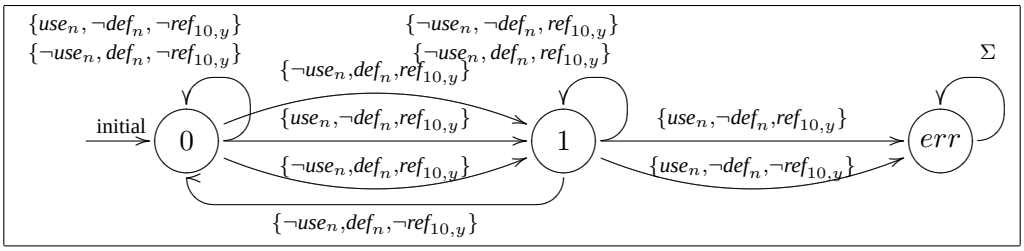


Figure 7.6: A heap safety automaton $A^{an}_{10,y,n}$ for assign null to $y.n$ at 10.

```
[1]  Node root = CreateTree();
[2]  processTree(root.right);
     ... // no further uses of root
```

Figure 7.7: A code snippet demonstrating the importance of assign-null analysis

(for a more formal treatment of semantic preserving transformations see [96]). As in the free property case, our assign-null property can also handle arbitrary reference expressions (e.g., of the form `exp.f`), by introducing a new program variable `z`, assigned with `exp`, and verifying the `z.f` may be issued just after the statement `z = exp`.

The above definition of the assign-null property directly and naturally corresponds to the ETL property

$$\Box\, \forall v.at[pt] \wedge x(v) \rightarrow \bigcirc \neg use_n(v)\, \mathcal{W}\, def_n(v) \tag{7.2}$$

In the formulation of this property, we use the combination of the predicate $at[pt]$ and the next temporal operator to achieve the same effect of using the $after[pt]$ predicate, as this exposes the temporal relationships in a manner closer to Definition 7.1.9.

The potential for space savings beyond GC is demonstrated using the code snippet in Fig. 7.7. A tree of objects is allocated, but only the right side of the tree is processed. We assume that the type `Node` contains two instance fields: `left` and `right`. After line 1 all tree objects are reachable, thus GC cannot reclaim the entire left subtree of the root. However, it is easy to see that the assign-null property $\langle 1, \texttt{root}, \texttt{left} \rangle$ holds, thus it is safe to insert a `root.left = null` statement after line 1 allowing GC to collect the left side of the tree before the processing at line 2.

Simultaneous Verification of Multiple Properties

So far we showed how to verify the free and assign-null properties for a single program point. Clearly, in practice one wishes to verify these properties for a set of program points without repeating the verification procedure for each program point. Our framework supports simultaneous verification of multiple properties, and in particular verification of properties for multiple program points. Assuming $\text{HSA}_1, \ldots, \text{HSA}_k$ describe k verification properties, then k automaton states $s_1, \ldots, s_k$ are maintained

```
public class SimultaneousVerification {
  public static void main(String args[]) {
    Object x1, ..., xk;
    Object y1, ..., yk;
    Random r = new Random();
    int count = r.nextInt();
    x1 = new Object();
pt1 y1 = x1;
    x2 = new Object();
pt2 y2 = x2;
    ⋮
    xk = new Object();
ptk yk = xk;
    if (count > 1) {
      y1 = x1;
    }
    if (count > 10) {
      y1 = x2;
    }
    ⋮
    if (count > 73) {
      y1 = xk;
    }
  }
}
```

Figure 7.8: A program demonstrating exponential blowup due to simultaneous verification of the free properties $\{\langle pt_i, x_i\rangle | 1 \leq i \leq k\}$.

for every program object, where s_i maintains an automaton state for HSA_i. Technically, as described in Section 7.1.3, a state s_i is represented by automaton state predicates $s_i[q]$, where q ranges over the states of HSA_i. The events associated with the automata $\text{HSA}_1, \ldots, \text{HSA}_k$ at a program point are triggered simultaneously, updating the corresponding automaton state predicates of individuals.

The worst-case cost of simultaneous verification of properties is higher than the worst-case cost of verifying the same properties one by one (see Chapter 6). For example, an attempt to simultaneously verify the free properties $\{\langle pt_i, x_i\rangle | 1 \leq i \leq k\}$ for the program of Fig. 7.8 exhibits an exponential blowup due to recording of the correlations between the various property (typestate) automata. For this example program, simultaneous verification for $k = 10$ takes 125 seconds and consumes 58.81MB of memory, whereas verification of a single property requires only 5.9 seconds and 2.72MB.

Nevertheless, verifying properties one by one ignores the potential of computing overlapping heap information just once, whereas in simultaneous verification of properties this overlap is taken into consideration. Thus, we believe that in practice simultaneous verification of a small number of properties

may sometimes achieve a lower cost than verifying the properties one by one. In fact, our initial findings in [95] show that verifying two properties one by one, takes close to double the time it takes to verify these properties simultaneously. This is because for a small number of properties, and a program performing intricate heap manipulations, verification cost is dominated by computation of heap information.

7.2 Automatically Verifying Concurrent Queue Algorithms

In this section, we show how the TVLA/3VMC framework can be applied to automatically verify partial correctness of non-trivial concurrent queue algorithms.

7.2.1 Concurrent Queue Algorithms

Concurrent FIFO queues are widely used in concurrent systems. Queues are used in scheduling mechanisms, and as the basis of many concurrent algorithms. Concurrent manipulation of a shared queue requires synchronization to guarantee consistent results. An ill-synchronized concurrent queue may be subject to read-write conflicts, write-write conflicts, or both.

A naive concurrent queue implementation uses a single shared lock to prevent concurrent manipulations of queue contents. Naturally, this limits the level of system concurrency. Many algorithms were suggested to increase concurrency while maintaining the correctness of queue manipulations [71, 100, 82, 111, 99]. The algorithms in [71, 100, 82, 99] are given without a formal proof of correctness, and [111] provides a manual formal proof.

We focus on the non-blocking queue and two-lock queue algorithms presented in [71]. A Java-like code for the queue implementations is given in Fig. 7.9.

To emulate the intention of [71], our programming model diverges from Java by assuming a free operation, and supporting several operations defined below.

In this section, we present the concurrent queue algorithms and the correctness properties we will verify for these algorithms.

Non-Blocking Queue

Java-like pseudo-code for the non-blocking queue algorithm is shown in Fig. 7.9(a). The queue uses an underlying singly-linked list which is pointed-to by two reference variables — Head and Tail, pointing to the head and tail of the queue correspondingly. The list always contains a dummy item at its head to avoid degenerate cases.

The algorithm is based on iterated attempts of a thread to perform a queue operation without being interrupted by other threads. A thread operates on shared-variables only using the compare-and-swap (CAS) primitive which allows it to atomically observe possible updates by other threads and apply its own update when the value of the shared variable has not been updated by other threads.

The CAS primitive takes 3 arguments — an address, an expected value, and a new value, it then atomically compares the address value to the expected value, and if the values are equal, it updates the address to contain the new value. If the address value is not equal to the expected value, no update is performed.

```
 class NonBlockingQueue {
  private QueueItem Head;
  private QueueItem Tail;
  ...
public NonBlockingQueue() {
  node = new QueueItem();
  node.next.ref = NULL;
  this.Head = this.Tail = node;
}
public void enqueue(Object value) {
e1   node = new QueueItem(value);
e2   node.value = value;
e3   node.next.ref = NULL;
e4   while(true) { //Keep trying until done
e5     tail = this.Tail;
e6     next = tail.ref.next;
e7     if (tail == this.Tail) {
e8       if (next.ref == NULL) {
e9         if CAS(tail.ref.next, next,
             <node, next.count+1>) {
e10          break // enqueue done
e11        }
e12      } else {
e13          CAS(this.Tail, tail,
               <next.ref, tail.count+1>);
e14      }
e15    }
e16  }
e17  CAS(this.Tail, tail, <node, tail.count+1>);
e18 }
public Object dequeue() {
    Object result = null;
d1   while(true) {
d2     head = this.Head;
d3     tail = this.Tail;
d4     next = head.next;
d5     if (head == this.Head) {
d6       if (head.ref == tail.ref) {
d7         if (next.ref == NULL) {//is empty?
d8           return result;
d9         }
d10        CAS(this.Tail, tail,
             <next.ref, tail.count+1>);
d11      } else { //No need to deal with Tail
d12        result = next.ref.value;
d13        if CAS(this.Head, head,
             <next.ref, head.count+1>) {
d14          break; // dequeue done
d15        }
d16      }
d17    }
d18  }
d19  free(head.ref);
d20  return result;
d21  }
```

(a)

```
// TwoLockQueue.java
class TwoLockQueue {
    private QueueItem head;
    private QueueItem tail;
    private Object headLock;
    private Object tailLock;
    ...
public TwoLockQueue() {
  node = new QueueItem();
  node.next = null;
  this.head = this.hail = node;
}

public void enqueue(Object value) {
lp1    QueueItem x_i =
         new QueueItem(value);
lp2    synchronize(tailLock) {
lp3        tail.next = x_i;
lp4        tail = x_i;
lp5    }
lp6  }

public Object dequeue() {
       Object x_d;
lt1    synchronized(headLock) {
lt2        QueueItem node = this.head;
lt3        QueueItem new_head =
             this.head.next;
lt4        if (new_head != null) {
lt5          x_d = new_head.value;
lt6          new_head = first;
lt7          new_head.value = null;
lt8          free(node);
           }
lt9    }
lt10   return x_d;
lt11 }
}
```

(b)

```
// QueueItem.java
class QueueItem {
    public QueueItem next;
    public Object value;
    ...
}
```

CAS-based algorithms may suffer from the "ABA" problem [71] in which a sequence of read-modify-CAS results with a swap when it should not. This happens when a thread t_1 reads a value A of a shared variable, computes a new value and preforms a CAS. Meanwhile, another thread t_2 changes the value of the shared variable from A to B and back to A. In order to avoid this problem, each reference variable is augmented with a modification counter and shared references are only updated through the CAS primitive which increments the value of the modification counter. This could have been modeled in Java by adding a wrapper class which contains a reference and an unsigned integer counter. To simplify the exposition of our figures, we have added a primitive type that consists of a reference-value `ref` and an integer value `count` for the modification counter. All reference operations that use only the reference name apply to both components, for example, the assignment at label e_5 assigns the values of `this.Tail.ref` and `this.Tail.count` to `tail.ref` and `tail.count` correspondingly. When we specifically update a single component of the reference variable, we state that explicitly as at label d_6 which performs a comparison of the `ref` component of two reference variables.

Two-Lock Queue

Fig. 7.9(b) shows a Java-like code for the two-lock queue algorithm. This algorithm also uses an underlying linked-list, and uses a dummy item at the list head to simplify special cases. The algorithm uses a separate head lock and tail lock to separate synchronization of enqueueing and dequeueing threads.

Correctness of Algorithms

The correctness of the queue algorithms in [71] is established by an informal proof. Safety of the algorithm is shown by induction, proving that the following properties are satisfied by the algorithm:

P1 The linked list is always connected.

P2 Nodes are only inserted after the last node of the linked list.

P3 Nodes are only deleted from the beginning of the linked list.

P4 *Head* always points to the first node in the linked list.

P5 *Tail* always points to a node in the linked list.

In the following sections, we formally state these claims, and automatically verify them using TVLA/3VMC.

7.2.2 Vanilla Verification Attempt

In this section, we describe the basic steps required to verify the concurrent queue algorithms using TVLA/3VMC.

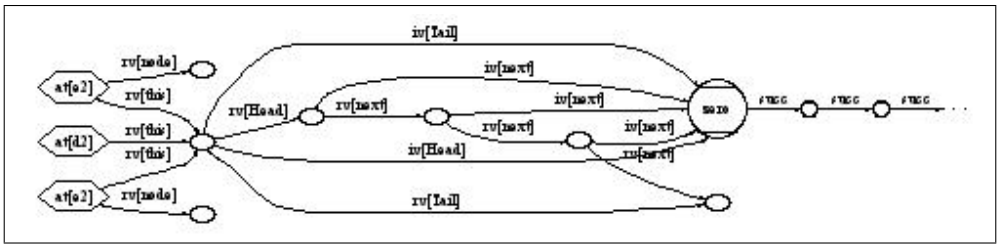


Figure 7.10: A concrete configuration $C^{\natural}_{7.10}$ with two enqueueing and one dequeueing threads.

Representing Program Configurations using First-Order Logical Structures

We now show how to apply our technique for verifying the concurrent queue algorithms.

The non-blocking queue algorithm uses unsigned integer values as reference time-stamps. As described in Section 2.3, we represent integer values using individuals of type unsigned integer, the unary predicate $zero(v)$, the binary predicate $succ(v_1, v_2)$, and the binary predicate $iv[fld](v_1, v_2)$. This allows us to naturally and quite precisely model an integer being incremented and decremented. It is also possible to support arbitrary arithmetic operations on integers, however, the abstraction presented in Section 7.2.3 is not precise enough to provide useful results when the verified property depends on the result of such operations.

To ease presentation, we depict nodes that represent unsigned integers as circles with straight margins.

Example 7.2.1 *The configuration $C^{\natural}_{7.10}$ shown in Fig. 7.10 corresponds to a global state of the non-blocking queue program with 3 threads: two enqueueing threads and a single dequeueing thread. The two enqueueing threads are at label e_2 and have just allocated new nodes to be enqueued. Each enqueueing thread refers to its node by its* `node` *field.*

All threads in the example use a single shared queue containing 4 items (including the dummy item). The integer values of the fields `Head` *and* `Tail` *in this configuration are both* 0.

Safety

The first step in verifying the properties of Section 7.2.1 in TVLA/3VMC is to formulate them in FO^{TC} using the predicates defined in Table 2.1. In Table 7.3 these formulae are given for the non-blocking queue algorithm. The formulation of these properties for the two-lock queue only differs in label names. For each property defined informally in Section 7.2.1, we provide a corresponding formula in FO^{TC}.

In the table, we use the shorthand nbq to abbreviate `NonBlockingQueue`.

Formula P1 uses transitive reachability from `Tail` and `Head` to require that each object that is reachable from the queue tail (including the tail node itself) is also reachable from the queue head — thus the queue is always connected. Note that requirement P5 guarantees that a tail element always

	Property	**Property Formula**
P1	tail reachable from head	$\forall q : nbq, v_t.rv[Tail](q, v_t)$ $\implies \exists v_h.rv[Head](q, v_h) \wedge rv[next]^*(v_h, v_t)$
P2	insert after last	$\forall q : nbq, t_i : thread, v_i, v_t.at[e_{18}](t_i) \wedge rv[node](t_i, v_i) \wedge rv[tail](t_i, v_t)$ $\wedge rv[this](t_i, q) \rightarrow rv[next](v_t, v_i) \wedge rv[Tail](q, v_i)$
P3	delete first	$\forall q : nbq, t_d : thread, v_d, v_h.at[d_{19}](t_d) \wedge rv[head](t_d, v_d)$ $\wedge rv[this](t_d, q) \wedge rv[Head](q, v_h) \implies rv[next](v_d, v_h)$
P4	head first	$\neg\exists q : nbq, v, u.rv[Head](q, v) \wedge rv[next](u, v)$
P5	tail exists	$\forall q : nbq.\exists v.rv[Tail](q, v)$

Table 7.3: Safety properties for non-blocking queue algorithm.

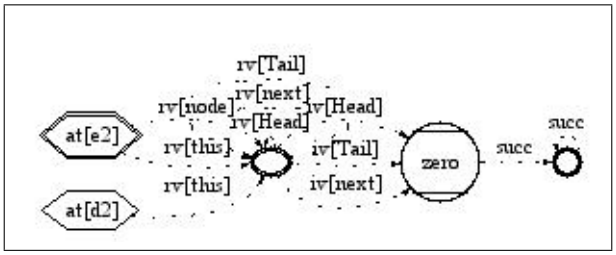


Figure 7.11: An abstract configuration $C_{7.10}$ representing the concrete configuration $C^{\natural}_{7.10}$ of Fig. 7.10.

exists. Formula P2 uses the (program) location predicate $at[e_{18}](t)$ in order to check the requirement only at the end of an insertion operation, when it is meaningful. In this formula, we treat the local variable node as a field of the thread object. Formula P3 similarly uses the location predicate $at[d_{19}](t)$ to bind the requirement with the end of a deletion operation. Formula P4 simply requires that there is no queue element u such that it precedes the head of the queue. Finally, formula P5 requires that a tail element exists.

Abstraction

Example 7.2.2 *The abstract configuration $C_{7.10}$ shown in Fig. 7.11 is obtained by applying canonical abstraction to the concrete configuration $C^{\natural}_{7.10}$ of Fig. 7.10.*

The summary thread-node represents the two enqueueing threads of the concrete configuration $C^{\natural}_{7.10}$, the summary unsigned-integer node (double-line circle with straight margins) summarizes all unsigned integers but zero, the third summary node summarizes all queue items, and the queue object itself.

Note that this abstract configuration represents an infinite number of configurations. For example, it represents any configuration in which an arbitrary number of enqueuing threads have just allocated new nodes to be enqueued, and are sharing the same queue with an arbitrary number of dequeueing

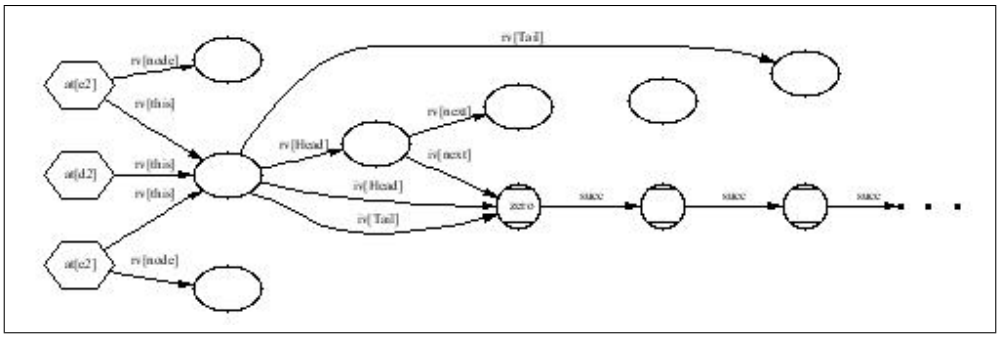


Figure 7.12: A concrete configuration $C^{\natural}_{7.10,1}$ that is embedded in $C_{7.10}$ and violates queue connectedness (property P1).

threads that are at their initial labels.

Unfortunately, this abstract configuration also represents the concrete configuration $C^{\natural}_{7.10,1}$ *which violates the connectedness property (P1), meaning that we fail to verify that P1 holds. Indeed, since each subformula of P1's body evaluates to* $1/2$ *over the abstract configuration* $C_{7.10}$*, using Kleene evaluation of boolean operators yields the value* $1/2$ *for P1. In the next section, we will see a way to remedy that.*

7.2.3 Refining the Vanilla Solution

In order to verify the desired properties, in this section we refine the abstraction to record essential information. A natural way to do that would be to record which property-formulae hold using nullary predicates. This is a useful technique, also known as predicate abstraction [48]. TVLA/3VMC also allows to use unary predicates in order to observe whether subformulae hold for a given individual. This allows TVLA/3VMC to provide useful results without changing the set of predicates for each program. We believe that the same distinctions can be used for many programs. Furthermore, these distinctions correspond to fundamental properties of data-structures (e.g., sharing, reachability). This section confirms this by showing that the standard set of distinctions suffices for verifying all the desired properties for the concurrent queue algorithms.

Technically, refining the abstraction is achieved by introducing the unary predicates of Table 7.4. The additional information recorded refines the abstraction and reduces the set of concrete configurations that are represented by an abstract configuration.

In principle, some instrumentation predicates could be derived automatically (e.g., [41]), however, for this case study we just use the standard TVLA/3VMC instrumentation predicates.

Predicates $rt[fld, n](t, o)$ (we use n as a shorthand for $next$ in the predicate name) allow us to track reachability information of items inside the queue. For example, the instrumentation predicate $rt[Head, n](v)$ may be used to track reachability of items from the head of the queue using a path of

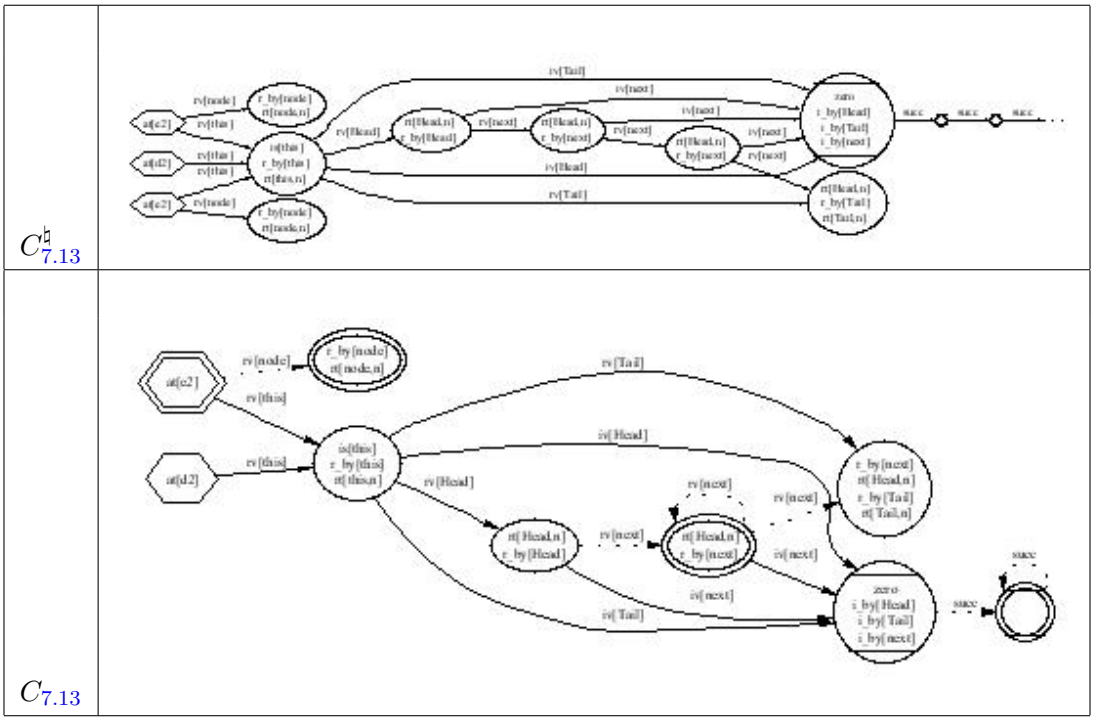


Figure 7.13: Concrete configuration $C^{\natural}_{7.13}$ using instrumentation predicates, and its canonical abstraction $C_{7.13}$.

Predicate	Intended Meaning	Defining Formula
$r\_by[fld](l)$	l is referenced by the field fld of some object	$\exists o.rv[fld](o,l)$
$i\_by[fld](n)$	n is the integer value of fld of some object	$\exists o.iv[fld](o,l)$
$is[fld](o)$	o is shared by fld of two different objects	$\exists v_1,v_2.\neg eq(v_1,v_2) \wedge rv[fld](v_1,o)$ $\wedge rv[fld](v_2,o)$
$exists[fld](o)$	there exists an object referenced by fld of o	$\exists v_1.rv[fld](o,v_1)$
$is\_acquired(l)$	l is acquired by some thread	$\exists t.held\_by(l,t)$
$rt[fld,n](o)$	o is reachable from object referenced by field fld using path of next fields	$\exists t,o_t.rv[fld](t,o_t)$ $\wedge\, rv[next]^*(o_t,o)$

Table 7.4: Instrumentation predicates used in our example program.

next references. These predicates are an adaptation for multithreaded programs of the reachability instrumentation predicates presented in [91]. Similarly, predicates $is[fld](o)$ are an adaptation of sharing predicates of [91]. The predicates $is\_acquired(l)$ and $r\_by[fld](l)$ were discussed in Section 2.4.3, and the predicates $exists[fld](o)$ used there but not explicitly mentioned. Since these predicates record widely-used *fundamental properties* of data-structures and thread/lock relationships, they are part of the standard predicates used in TVLA/3VMC.

Subformulae of the safety properties are replaced with the corresponding instrumentation predicate to improve precision.

Example 7.2.3 *Fig. 7.13 shows the concrete configuration $C^{\natural}_{7.13}$ which is an instrumented version of $C^{\natural}_{7.10}$, and its canonical abstraction $C_{7.13}$. The additional information recorded by the instrumentation predicates $rt[Head, n](v)$ and $rt[Tail, n](v)$ allows us to observe that queue connectedness (property P1) is maintained in the abstract configuration $C_{7.13}$ since P1 evaluates to* 1*. Moreover, this implies that concrete configurations of the form of $C^{\natural}_{7.10,1}$ are no longer represented.*

7.2.4 Experimental Results

Our prototype implementation operates directly on abstract configurations using *abstract transformers*, thereby obtaining actions which are more conservative than the ones obtained by the best transformers. Our experience shows that the abstract transformers used in the implementation are still precise enough to allow verification of our safety properties.

Update formulae for the instrumentation predicates used in this case study were supplied manually due to technical limitations of automatic derivation using finite differencing [86].

Table 7.5 presents the analysis results for various variations of the concurrent queue algorithms.

For the non-blocking queue, we have also tested a version in which the conditional in label e_8 is flipped, i.e., it checks for the next field being non-equal to null. As another erroneous version, we have used an uninitialized queue in which no dummy node was present. Both cases reported errors.

For the two-lock queue, we have also tested a version in which no synchronization is imposed on producer threads inserting items into the queue. In this version, we show that it is possible for requirement 1 to be violated, and the underlying linked-list to be broken.

Limitations: Since our tool does not apply any partial-order reductions and does not attempt to decrease the level of interleaving, it is currently limited to small concurrent programs or to ones that are well-synchronized. This is due to the worst-case complexity of our algorithm which is doubly exponential in the number of labels.

A fundamental question in program analysis is how to predict the precision of a given analysis on a given program. In principle, this is a hard question, we note that the abstraction in TVLA/3VMC significantly loses information when arbitrary arithmetic operations on integer variables (which affect

Program	**Configs**	**Space (MB)**	**Time (sec)**	**Comments**
nbq_enqueue	1833	14.2	727	unbounded number of enqueue-ing threads
nbq_dequeue	1098	5.3	309	unbounded number of dequeue-ing threads
nonblockq_err1	36	0.1	11	err - negated condition at e8
nonblockq_uni	17	0.1	3	err - start with uninitialized queue
tlq_enqueue	982	10	6162	unbounded number of enqueueing thrads
tlq_dequeue	225	4.1	304	unbounded number of dequeuing threads
twolockqn	975	7.5	577	single producer and single consumer
twolockq_err1	24	0.1	30	err - broken producer synchronization

Table 7.5: Analysis results for variations of the queue algorithms — number of configurations explored, space requirements, and analysis time.

the safety of the algorithm) are performed.

7.3 Solving the Apprentice Challenge

In this section, we describe how our framework is applied for solving a Java verification challenge known as the Apprentice Challenge.

7.3.1 Problem Statement

The apprentice challenge was presented by Moore [73] as a challenge in verification of Java programs. The challenge is to show that the value of the `counter` variable of the `Container` class in Fig. 7.14 increases monotonically (under all possible schedules).

7.3.2 Solution

Our solution of the apprentice challenge does not assume any *a priori* bound on the number of `Job` threads or on the value of the `counter` field. This should be contrasted with previous attempts to solve the apprentice challenge using model-checking (i.e., the "finite Apprentice").

In our solution, we use the predicates described earlier in Section 2.3 and Section 7.2.3. The model used here could be easily extended to handle the overflow of integer variables (by introducing a special terminating node in the representation of the integers). For simplicity, we do not introduce any treatment of such overflow and assume that integers may increase infinitely.

The initial configuration for the apprentice challenge is shown in Fig. 7.15. In this configuration there is a single thread node, corresponding to the main program thread. This thread resides at the initial label $\texttt{gl}_1$, and is ready to be scheduled. The other nodes in this configuration represent integer values: one node represents the value zero, and the summary node summarizes the rest of the integer values.

Our system requires two technical modifications of the `incr()` method (shown in Fig. 7.16): (i) splitting the increment statement into two assignments, one assigning `counter + 1` to a temporary variable, and another copying the value of the temporary variable into `counter` (In principle, this could be performed by a trivial front-end); (ii) instrumenting the method to record the previous value of the counter, this again is a technical issue that could be avoided in principle. A conceptual view of the instrumented method is shown in Fig. 7.16. It is important to note that `prevcounter` is introduced as an additional predicate in the model and not as an additional program variable, i.e., it cannot be modified by the program.

7.3.3 Results

We applied 3VMC to verify that the original Apprentice program satisfies the goal property. Verification produced 1757 configurations and took approximately 120 seconds and 2.46 MB of memory.

```
class Container {
   public int counter;
}

class Job extends Thread {
   Container objref;
   public Job incr () {
      synchronized(objref) {
         objref.counter = objref.counter + 1;
      }
      return this;
   }
   public void setref(Container o) {
      objref = o;
   }
   public void run() {
      for (;;) {
         incr();
      }
   }
}

class Apprentice {
   public static void main(String[] args) {
      Container container = new Container();
      for (;;) {
         Job job = new Job();
         job.setref(container);
         job.start();
      }
   }
}
```

Figure 7.14: Source of the Apprentice Challenge.

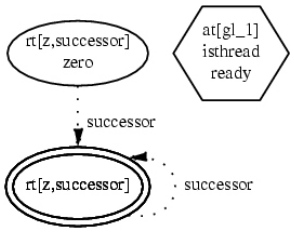


Figure 7.15: Initial configuration for the apprentice challenge.

```
public Job incr () {
   synchronized(objref) {
      //objref.prevcounter = objref.counter;
      temp = objref.counter + 1;
      objref.counter = temp;
   }
   return this;
}
```

Figure 7.16: Conceptual rewrite of `incr()` method.

We have also applied 3VMC to find errors in an erroneous version of the Apprentice program in which no synchronization was used by `Job` threads while performing the `incr()` operation. In this analysis, an error was detected after approximately 720 seconds, processing 6066 configurations taking 13.8 MB of memory. Note that since no synchronization was applied between `Job` threads, the number of possible interleaving considered in this exploration is huge.

Unlike the ACL2 solution for the apprentice challenge, our approach is based on a conservative abstraction of the concrete Java semantics. Generally, this means that we might produce alarms even when a property does hold for the verified program. However, for the Apprentice challenge, we are able to verify the goal property without any false alarms.

Chapter 8

Conclusions and Further Work

8.1 Conclusion

We have presented a parametric framework for specifying and verifying properties of concurrent and sequential heap-manipulating programs. Our framework generalizes existing model-checking techniques. The framework allows verification of multithreaded programs manipulating heap-allocated objects, and does not put a bound on the number of allocated objects (and threads).

The framework uses an integrated verification and pointer analysis, leading to results that are always more precise than those of the two-phased approach applied by other systems. In addition, our framework also handles properties of correlated objects.

Our framework combines thread scheduling information and information about the shape of the heap. This leads to error-detection algorithms that are more precise than existing techniques. Using this approach, we were able to automatically verify non-trivial properties of heap-manipulating programs that have not been automatically verified in the past.

We have also presented a technique for scaling verification to large(r) programs with a small number of false alarms. This allows us automatic verification of programs and properties not automatically verified earlier.

We have applied our framework to verify several interesting properties and programs, and in particular for applying compile-time GC and proving the correctness of concurrent queue algorithms.

8.2 Contrast with Closely Related Work

It is important to view our contributions in the context of closely related work. Fig. 8.1 shows a classification of our contributions (shown in bold typeface) and closely related work. Classification is shown using a 3-dimensional cube, as done in Chapter 1, and using the same dimensions:

Heap Abstraction Describes the strength of the applied heap-abstraction. Zero on this axis means that

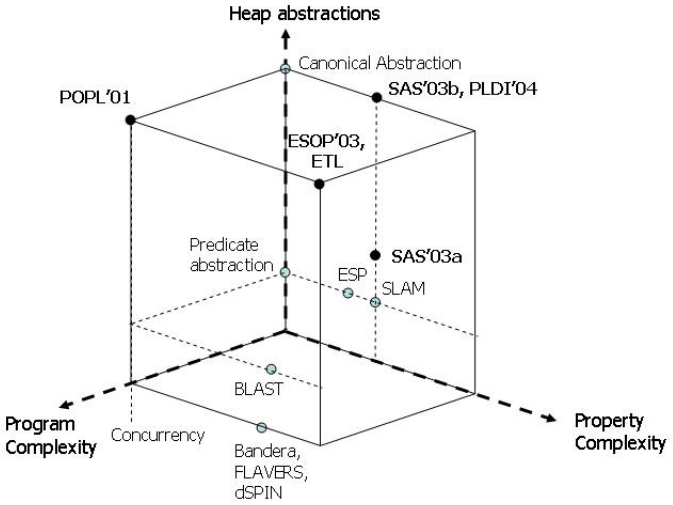


Figure 8.1: Overview of closely related work.

no heap abstraction is used, thus forcing an assumed a priori bound on the number of allocated objects and threads.

Program Complexity Describes the complexity of the programs that could be handled. Along this dimension we only distinguish between sequential and concurrent programs.

Property Complexity Describes the complexity of the properties that could be handled. Property complexity ranges from non-temporal safety properties to full temporal specification.

Bandera [19], FLAVERS [77], and dSPIN [32] do not apply any heap abstraction. These approaches are therefore forced to assume an a priori bound on the number of allocated objects and threads, making them generally unsound.

SLAM [72], and BLAST [53] take the two-phased approach, and use a predicate abstraction to verify a safety property against a finite-state model of a program. The finite-state model is produced using a preceding pointer-analysis phase. ESP [27], uses a two-phased approach for verifying typestate properties. As mentioned earlier, the two-phased approach may result in an extensive number of false alarms, but is more scalable since the pointer-analysis phase may be flow-insensitive.

8.3 Further Work

8.3.1 Property Guided Abstraction

Ideally, we would like the cost of verification to depend only on the verified program and the complexity of the verified property.

I plan to further investigate methods for directing the abstraction by the property specification provided by the user. More generally, users often have some insight of what actually makes their program work correctly. Rather than forcing the user to write program annotations and loop-invariants which are often complicated and non-intuitive, it would be interesting to let the user direct the abstraction used by the static-analysis algorithms. A first attempt to let user specification direct the abstraction was described in Chapter 6. In the approach presented there, the user provides a strategy for choosing "relevant" heap-allocated objects that should be abstracted using a more precise abstraction than the rest of the heap.

8.3.2 Verification of Heap-Manipulating Programs

The verification algorithms we have investigated so far are very precise and appealing but are not likely to scale to verification of industrial software. Verifying real-world applications requires cheaper verification algorithms. There are several directions I would like to pursue here:

- use property-guided abstraction and apply precise (and costly) abstraction only to some parts of the heap.
- develop efficient verification algorithms for useful subsets of ETL specifications.
- apply partial-order reductions to explore only representatives of equivalent interleavings.
- use information from dynamic (runtime) analyses to direct static verification.

Bibliography

[1] P. A. Abdulla, A. Annichini, S. Bensalem, and A. Bouajjani. Verification of infinite-state systems by combining abstraction and reachability analysis. *Lecture Notes in Computer Science*, 1633, 1999.

[2] O. Agesen, D. Detlefs, and E. Moss. Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 269–279. ACM Press, June 1998.

[3] J. Aldrich, C. Chambers, E.G. Sirer, and S. Eggers. Static analyses for eliminating unnecessary synchronization from Java programs. In Agostino Cortesi and Gilberto Filé, editors, *Static Analysis*, volume 1694 of *Lecture Notes in Computer Science*, pages 19–38. Springer, 1999.

[4] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *Proc. IEEE Symp. on Security and Privacy*, Oakland, CA, May 2002.

[5] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *Proc. Conf. on Prog. Lang. Design and Impl.*, pages 203–213, June 2001.

[6] T. Ball and S.K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001*, LNCS 2057, pages 103–122, 2001.

[7] T. Ball and S.K. Rajamani. SLIC: A Specification Language for Interface Checking (of C). Technical Report MSR-TR-2001-21, MSR, 2001.

[8] S. Bensalem, V. Ganesh, Y. Lakhnech, C. Muñoz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saïdi, N. Shankar, E. Singerman, and A. Tiwari. An overview of SAL. In *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, June 2000.

[9] S. Bensalem, Y. Lakhnech, and S. Owre. InVeSt: A tool for the verification of invariants. *LNCS*, 1427, 1998.

[10] R. Bodik, R. Gupta, and M.L. Soffa. Refining data flow information using infeasible paths. In *Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT interna-*

tional symposium on Foundations of software engineering, pages 361–377. Springer-Verlag New York, Inc., 1997.

[11] P.A. Buhr, M. Fortier, and M.H. Coffin. Monitor classification. *ACM Computing Surveys*, 27(1):63–107, March 1995.

[12] L. Cardelli and A.D.Gordon. Mobile ambients. In *Proc. FoSSaCS'98, vol. 1378 of LNCS*, pages 140–155. Springer, 1998.

[13] D.R. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *Proc. Conf. on Prog. Lang. Design and Impl.*, pages 296–310, New York, NY, 1990. ACM Press.

[14] E. M. Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks. *Trans. on Prog. Lang. and Syst.*, 19(5):726–750, September 1997.

[15] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV'00*, July 2000.

[16] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *Trans. on Prog. Lang. and Syst.*, 16(5):1512–1542, September 1994.

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

[18] J. Corbett. Using shape analysis to reduce finite-state models of concurrent java programs. October 1998.

[19] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, R. Shawn, and L. Hongjun. Bandera: Extracting finite-state models from Java source code. In *Proc. 22nd ICSE*, June 2000.

[20] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. A language framework for expressing checkable properties of dynamic software. In *SPIN*, 2000.

[21] James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. Intl. Conf. on Software Eng.*, pages 439–448, June 2000.

[22] J.C. Corbett, M.B. Dwyer, J. Hatcliff, and Robby. Expressing checkable properties of dynamic systems: the bandera specification language. *STTT*, 4(1):34–56, October 2002.

[23] B. Courcelle. On the expression of graph properties in some fragments of monadic second-order logic. In N. Immerman and P.G. Kolaitis, editors, *Descriptive Complexity and Finite Models: Proceedings of a DIAMCS Workshop*, chapter 2, pages 33–57. American Mathematical Society, 1996.

[24] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Symp. on Princ. of Prog. Lang.*, pages 238–252, New York, NY, 1977. ACM Press.

[25] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. Symp. on Principles of Prog. Languages*, pages 269–282, New York, NY, 1979. ACM Press.

[26] P. Cousot and R. Cousot. Temporal abstract interpretation. In *Proc. of 27th POPL*, pages 12–25, January 2000.

[27] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proc. Conf. on Prog. Lang. Design and Impl.*, pages 57–68, June 2002.

[28] S. Das, D.L. Dill, and S. Park. Experience with predicate abstraction. In *11th Int. Conf. on Computer-Aided Verification*. Springer-Verlag, July 1999. Trento, Italy.

[29] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. Conf. on Prog. Lang. Design and Impl.*, pages 59–69, June 2001.

[30] R. DeLine and M. Fähndrich. Adoption and focus: Practical linear types for imperative programming. In *Proc. Conf. on Prog. Lang. Design and Impl.*, pages 13–24, June 2002.

[31] C. Demartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent Java programs. *Software: Practice and Experience*, 29(7):577–603, June 1999.

[32] C. Demartini, R. Iosif, and R. Sisto. dSPIN : A dynamic extension of SPIN, September 1999.

[33] E. W. Dijkstra. *A Discipline of programing*. Prentice-Hall, 1976.

[34] D. Distefano. *On Model Checking the Dynamics of Object-Based Software*. PhD thesis, Twente University, 2003.

[35] N. Dor, M. Rodeh, and M. Sagiv. Checking cleanness in linked lists. In *SAS'00, Static Analysis Symposium*. Springer, 2000. Available at "http://www.math.tau.ac.il/∼ nurr".

[36] N. Dor, M. Rodeh, and M. Sagiv. CSSV: towards a realistic tool for statically detecting all buffer overflows in C. *ACM SIGPLAN Notices*, 38(5):155–167, May 2003.

[37] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proc. of Int. Conf. on Software Engineering*, pages 411–421, May 1999.

[38] E. Emerson and A. P. Sistla. Symmetry and model checking. In *Proc. 5th Workshop on Computer-Aided Verificaton*, June/July 1993.

[39] M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, New York, NY, 1994. ACM Press.

[40] J. Field, D. Goyal, G. Ramalingam, and E. Yahav. Shallow finite state verification. Technical Report RC22673, IBM T.J. Watson Research Center, December 2002.

[41] J. Field, D. Goyal, G. Ramalingam, and E. Yahav. Typestate verification: Abstraction techniques and complexity results. In *Proc. of SAS'03*, volume 2694 of *LNCS*, pages 439–462. Springer, June 2003.

[42] M. Fitting and R.L. Mendelsohn. *First-Order Modal Logic*, volume 277 of *Synthese Library*. Kluwer Academic Publishers, Dordrecht, 1998.

[43] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended static checking for java. In *Proc. Conf. on Prog. Lang. Design and Impl.*, pages 234–245, Berlin, June 2002.

[44] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proc. Conf. on Prog. Lang. Design and Impl.*, pages 1–12, Berlin, June 2002.

[45] G.E. Hughes and M.J. Creswel. *An Introduction to Modal Logic*. Methuen, London, 1982.

[46] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *Journal of the ACM*, 47(2):361–416, 2000.

[47] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, 1997.

[48] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. *LNCS*, 1254:72–83, 1997.

[49] Suzanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In *In Proceedings of the 9th Conference on Computer-Aided Verification (CAV'97)*, pages 72–83, Haifa, Israel, June 1997.

[50] S.Z. Guyer and C. Lin. Client-driven pointer analysis. In *Proc. of SAS'03*, volume 2694 of *LNCS*, pages 214–236, June 2003.

[51] P.B. Hansen. Java's insecure parallelism. *ACM SIGPLAN Notices*, 34(4):38–45, April 1999.

[52] K. Havelund and T. Pressburger. Model checking Java programs using Java pathfinder. *Int. J.on Soft. Tools for Technology Transfer*, 2(4), April 2000.

[53] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002.

[54] L. H. Holley and B. K. Rosen. Qualified data flow problems. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 68–82. ACM SIGACT and SIGPLAN, ACM Press, 1980.

[55] G. J. Holzmann. Proving properties of concurrent systems with SPIN. In *Proc. of the 6th Int. Conf. on Concurrency Theory (CONCUR'95)*, volume 962 of *LNCS*, pages 453–455, Berlin, GER, August 1995. Springer.

[56] Katsuro Inoue, Hiroyuki Seki, and Hikaru Yagi. Analysis of functional programs to detect run-time garbage cells. *Trans. on Prog. Lang. and Syst.*, 10(4):555–578, October 1988.

[57] Y. Kesten and A. Pnueli. Verification by augmented finitary abstraction. *INFCTRL: Information and Computation (formerly Information and Control)*, 163, 2000.

[58] Y. Kesten, A. Pnueli, and M. Vardi. Verification by augmented abstraction: The automata-theoretic view. *JCSS: J. of Comp. Sys. Sci.*, 62, 2001.

[59] A. Knapp, P. Cenciarelli, B. Reus, and M. Wirsing. An event-based structural operational semantics of multi-threaded java, 1998.

[60] V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *Proc. Symp. on Principles of Prog. Languages*, January 2002.

[61] W. Landi and B. G. Ryder. Pointer-induced aliasing: A problem classification. In *Proc. Symp. on Principles of Prog. Languages*, pages 93–103, New York, NY, 1991. ACM Press.

[62] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992.

[63] D. Lea. *Concurrent Programming in Java*. Addison-Wesley, Reading, Massachusetts, 1997.

[64] T. Lev-Ami and M. Sagiv. TVLA: A framework for Kleene based static analysis. In *SAS'00, Static Analysis Symposium*. Springer, 2000. Available at http://www.math.tau.ac.il/∼tla.

[65] T. Lev-Ami and M. Sagiv. TVLA: A framework for Kleene based static analysis. In *Proc. Static Analysis Symp.*, volume 1824 of *LNCS*, pages 280–301. Springer-Verlag, 2000.

[66] D. Lewis. Counterpart theory and quantified modal logic. *Journal of Philosophy*, LXV(5):113–126, 1968.

[67] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, January 1997.

[68] Z. Manna and A. Pnueli. Completing the temporal picture. *Theoretical Computer Science*, 83(1):97–130, June 1991.

[69] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer, 1995.

[70] K. L. McMillan. Verification of infinite state systems by compositional model checking. In *Proc. of CHARME '99*, volume 1703 of *LNCS*, pages 219–237, 1999.

[71] M.M. Michael and M.L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC '96)*, pages 267–275, New York, USA, May 1996. ACM.

[72] Microsoft Research. The SLAM project. http://research.microsoft.com/slam/, 2001.

[73] J. S. Moore and G. Porter. The apprentice challenge. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):193–216, 2002.

[74] R. Muth and S. Debray. On the complexity of flow-sensitive dataflow analyses. In *Proc. Symp. on Principles of Prog. Languages*, pages 67–80, New York, NY, 2000. ACM Press.

[75] N. Francez. *Fairness*. Springer-Verlag, New York, 1987.

[76] G. Naumovich, G.S. Avrunin, and L.A. Clarke. Data flow analysis for checking properties of concurrent Java programs. In *Proc. of the 1999 Int. Conf. on Soft. Eng.*, pages 399–410. IEEE Computer Society Press / ACM Press, 1999.

[77] G. Naumovich, L.A. Clarke, L.J. Osterweil, and M.B. Dwyer. Verification of concurrent sofware with FLAVERS. In *Proc. Intl. Conf. on Software Eng.*, pages 594–597, May 1997.

[78] R.H.B. Netzer and B.P. Miller. What are race conditions? some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, March 1992.

[79] F. Nielson, H. Riis Nielson, and M. Sagiv. A Kleene analysis of mobile ambients. In *Proceedings of ESOP'2000*, 2000.

[80] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 2001.

[81] A. Pnueli, J. Xu, and L.D. Zuck. Liveness with (0, 1, infty)-counter abstraction. In *Proceedings of the 14th International Conference on Computer Aided Verification*, pages 107–122. Springer-Verlag, 2002.

[82] S. Prakash, Y. Lee, and T. Johnson. A non-blocking algorithm for shared queues using Compare-and-Swap. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages 68–75, 1991.

[83] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, 16(5):1467–1471, 1994.

[84] G. Ramalingam, A. Warshavsky, J. Field, D. Goyal, and M. Sagiv. Deriving specialized program analyses for certifying component-client conformance. In *Proc. Conf. on Prog. Lang. Design and Impl.*, volume 37, 5, pages 83–94, June 2002.

[85] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proc. Symp. on Principles of Prog. Languages*, pages 49–61, 1995.

[86] T. Reps, M. Sagiv, and A. Loginov. Finite differencing of logical formulas for static analysis. In *In Proc. European Symp. on Programming*, 2003.

[87] M. Rinard. Analysis of multithreaded programs. *Lecture Notes in Computer Science*, 2126, 2001.

[88] N. Rinetskey and M. Sagiv. Interprocedural shape analysis for recursive programs. In R. Wilhelm, editor, *Proc. Intl. Conf. on Compiler Construction*, volume 2027 of *LNCS*, pages 133–149. Springer-Verlag, 2001.

[89] N. Rinetzky and M. Sagiv. Interprocedural shape analysis for recursive programs. *LNCS*, 2027:133–149, 2001.

[90] Robby, E. Rodriguez, M.B. Dwyer, and J. Hatcliff. Checking strong specifications using an extensible software model checking framework. In *Proc. of the int. conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *LNCS*, pages 404–420. Springer, 2004.

[91] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):217–298, 2002.

[92] H. Saidi. Model checking guided abstraction and analysis. In *Proceedings of the 7th International Static Analysis Symposium (SAS '00)*, 2000.

[93] R. Shaham, E.K. Kolodner, and M. Sagiv. Automatic removal of array memory leaks in Java. In *Int. Conf. on Comp. Construct.*, volume 1781 of *Lec. Notes in Comp. Sci.*, pages 50–66. Springer-Verlag, April 2000.

[94] R. Shaham, E.K. Kolodner, and M. Sagiv. Estimating the impact of heap liveness information on space consumption in Java. In *Int. Symp. on Memory Management*, pages 171–182. ACM, June 2002.

[95] R. Shaham, E. Yahav, E. K. Kolodner, and M. Sagiv. Establishing local temporal heap safety properties with applications to compile-time memory management. In *Proc. of the 10th International Static Analysis Symposium, SAS 2003*, volume 2694 of *LNCS*, June 2003.

[96] Ran Shaham. *Heap-Liveness-based Memory Management: Potential, Tools, and Algorithms*. PhD thesis, Tel Aviv University, 2004.

[97] A. Silberschatz and P. B. Galvin. *Operating Systems Concepts*. Addison-Wesley, Reading, 4 edition, 1994.

[98] S.D. Stoller. Model-checking multi-threaded distributed Java programs. In *Proc. 7th Int. SPIN Workshop on Model Checking of Software*, volume 1885 of *LNCS*, pages 224–244. Springer-Verlag, August 2000.

[99] J. M. Stone. A simple and correct shared-queue algorithm using Compare-and-Swap. In *Proceedings of Supercomputing '90*, pages 495–504, 1990.

[100] J. M. Stone. A non-blocking Compare-and-Swap algorithm for a shared circular queue. In S. Tzafestas et al., editors, *Parallel and Distributed Computing in Engineering Systems*, pages 147–152. Elsevier Science Publishers, 1992.

[101] R. E. Strom. Mechanisms for compile-time enforcement of security. In *Proc. of the 10th Annual ACM Symposium on Principles of Programming Lanuages*, pages 276–284, Austin, TX, January 1983.

[102] R.E. Strom and D.M. Yellin. Extending typestate checking using conditional liveness analysis. *IEEE Trans. Software Eng.*, 19(5):478–485, May 1993.

[103] R.E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986.

[104] C. Ungureanu and S. Jagannathan. Concurrency analysis for java. In *Proceedings of the 7th International Static Analysis Symposium (SAS '00)*, 2000.

[105] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a java optimization framework. In *Proc. of CASCON 1999*, pages 125–135, 1999.

[106] M. Vardi. An automata-theoretic approach to linear temporal logic. In *Proceedings of Banff'94*, 1994.

[107] M.Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 15 November 1994.

[108] A. Vermeulen. Java deadlock: The woes of multithreaded design. *Dr. Dobb's Journal of Software Tools*, 22(9):52, 54–56, 88, 89, September 1997.

[109] S. White, M. Fisher, R. Cattell, G. Hamilton, and M. Hapner. *JDBC API tutorial and reference*. Addison-Wesley, 1999.

[110] P. R. Wilson. Uniprocessor garbage collection techniques. In *Memory Management, International Workshop IWMM*, volume 637 of *Lec. Notes in Comp. Sci.*, pages 1–42. Springer-Verlag, September 1992.

[111] J. M. Wing and C. Gong. A library of concurrent objects and their proofs of correctness. Technical Report CMU–CS–90–151, CMU, 1990.

[112] E. Yahav. http://www.cs.tau.ac.il/∼yahave.

[113] E. Yahav. 3VMC user's manual, 2000. Available at http://www.math.tau.ac.il/∼yahave.

[114] E. Yahav. Solving the apprentice challenge using 3VMC, 2000. Available at http://www.cs.tau.ac.il/∼yahave.

[115] E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *Proc. Symp. on Principles of Prog. Languages*, pages 27–40, 2001.

[116] E. Yahav, A. Pnueli, T. Reps, and M. Sagiv. Efficient verification of temporal heap properties. Technical Report 339/04, Tel Aviv University, December 2003. *Submitted for publication*.

[117] E. Yahav and G. Ramalingam. Verifying safety properties using separation and heterogeneous abstractions. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 25–34. ACM Press, 2004.

[118] E. Yahav, T. Reps, and M. Sagiv. LTL model checking for systems with unbounded number of dynamically created threads and objects. Technical Report TR-1424, Computer Sciences Department, University of Wisconsin, Madison, WI, March 2001.

[119] E. Yahav, T. Reps, M. Sagiv, and R. Wilhelm. Verifying temporal heap properties specified via evolution logic. In *Proc. of the 12th European Symposium on Programming, ESOP 2003*, volume 2618 of *LNCS*, April 2003.

[120] E. Yahav and M. Sagiv. Automatically verifying concurrent queue algorithms. In Byron Cook, Scott Stoller, and Willem Visser, editors, *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2003.

Appendix A

2 and 3-valued FO^{TC}

In this appendix, we give a brief summary of 2 and 3 valued FO^{TC}. The material presented here is fairly standard and included only for completeness of presentation.

A.1 Syntax

Formally, the syntax of first-order formulae with transitive closure is defined as follows:

Definition A.1.1 *A* **formula** *over the* **vocabulary** $\mathcal{P} = \{eq, p_1, \ldots, p_n\}$ *is defined inductively, as follows:*

Atomic Formulae *The* **logical literals** 0 *and* 1 *are atomic formulae with no free variables.*

For every predicate symbol $p \in \mathcal{P}$ of arity k, $p(v_1, \ldots, v_k)$ is an atomic formula with free variables $\{v_1, \ldots, v_k\}$.

Logical Connectives *If φ_1 and φ_2 are formulae whose sets of free variables are V_1 and V_2, respectively, then $(\varphi_1 \wedge \varphi_2)$, $(\varphi_1 \vee \varphi_2)$, and $(\neg\varphi_1)$ are formulae with free variables $V_1 \cup V_2$, $V_1 \cup V_2$, and V_1, respectively.*

Quantifiers *If φ_1 is a formula with free variables $\{v_1, v_2, \ldots, v_k\}$, then $(\exists v_1 : \varphi_1)$ and $(\forall v_1 : \varphi_1)$ are both formulae with free variables $\{v_2, v_3, \ldots, v_k\}$.*

Transitive Closure *If φ_1 is a formula with free variables V such that $v_3, v_4 \notin V$, then $(TC\, v_1 \colon v_2)(\varphi_1)v_3v_4$ is a formula with free variables $(V - \{v_1, v_2\}) \cup \{v_3, v_4\}$.*

A formula is **closed** *when it has no free variables.*

A.2 2-valued Interpretation

In this section, we define the (2-valued) semantics for first-order logic with transitive closure in the standard way.

Definition A.2.1 *A* **2-valued interpretation** *of the language of formulae over $\mathcal{P}$ is a* **2-valued logical structure** *$S = \langle U^S, \iota^S \rangle$, where U^S is a set of* **individuals** *and ι^S maps each predicate symbol p of arity k to a truth-valued function:*

$$\iota^S(p)\colon (U^S)^k \to \{0,1\}.$$

An **assignment** *Z is a function that maps free variables to individuals (i.e., an assignment has the functionality $Z\colon \{v_1, v_2, \ldots\} \to U^S$). An assignment that is defined on all free variables of a formula φ is called* **complete** *for φ. In the sequel, we assume that every assignment Z that arises in connection with the discussion of some formula φ is complete for φ.*

The **(2-valued) meaning** *of a formula φ, denoted by $[\![\varphi]\!]_2^S(Z)$, yields a truth value in $\{0,1\}$. The meaning of φ is defined inductively as follows:*

Atomic Formulae *For an atomic formula consisting of a logical literal $\boldsymbol{l} \in \{0,1\}$, $[\![\boldsymbol{l}]\!]_2^S(Z) = l$ (where $l \in \{0,1\}$).*

For an atomic formula of the form $p(v_1, \ldots, v_k)$,

$$[\![p(v_1, \ldots, v_k)]\!]_2^S(Z) = \iota^S(p)(Z(v_1), \ldots, Z(v_k))$$

Logical Connectives *When φ is a formula built from subformulae φ_1 and φ_2,*

$$\begin{aligned} [\![\varphi_1 \wedge \varphi_2]\!]_2^S(Z) &= \min([\![\varphi_1]\!]_2^S(Z), [\![\varphi_2]\!]_2^S(Z)) \\ [\![\varphi_1 \vee \varphi_2]\!]_2^S(Z) &= \max([\![\varphi_1]\!]_2^S(Z), [\![\varphi_2]\!]_2^S(Z)) \\ [\![\neg\varphi_1]\!]_2^S(Z) &= 1 - [\![\varphi_1]\!]_2^S(Z) \end{aligned}$$

Quantifiers *When φ is a formula that has a quantifier as the outermost operator,*

$$\begin{aligned} [\![\forall v_1 : \varphi_1]\!]_2^S(Z) &= \min_{u \in U^S} [\![\varphi_1]\!]_2^S(Z[v_1 \mapsto u]) \\ [\![\exists v_1 : \varphi_1]\!]_2^S(Z) &= \max_{u \in U^S} [\![\varphi_1]\!]_2^S(Z[v_1 \mapsto u]) \end{aligned}$$

Transitive Closure *When φ is a formula of the form $(TC\ v_1\colon v_2)(\varphi_1)v_3v_4$,*

$$[\![(TC\ v_1\colon v_2)(\varphi_1)v_3v_4]\!]_2^S(Z) = \max_{\substack{n \geq 1, u_1, \ldots, u_{n+1} \in U, \\ Z(v_3) = u_1, Z(v_4) = u_{n+1}}} \min_{i=1}^{n} [\![\varphi_1]\!]_2^S(Z[v_1 \mapsto u_i, v_2 \mapsto u_{i+1}])$$

We say that S and Z **satisfy** *φ (denoted by $S, Z \models \varphi$) if $[\![\varphi]\!]_3^S(Z) = 1$. We write $S \models \varphi$ if for every Z we have $S, Z \models \varphi$.*

A.3 3-valued Interpretation

We now generalize Defn. A.2.1 to define the meaning of a formula with respect to a 3-valued structure.

Definition A.3.1 *A* **3-valued interpretation** *of the language of formulae over $\mathcal{P}$ is a* **3-valued logical structure** *$S = \langle U^S, \iota^S \rangle$, where U^S is a set of individuals and ι^S maps each predicate symbol p of arity k to a truth-valued function:*

$$\iota^S(p) \colon (U^S)^k \to \{0, 1, 1/2\}.$$

For an assignment Z, the **(3-valued) meaning** *of a formula φ, denoted by $[\![\varphi]\!]_3^S(Z)$, now yields a truth value in $\{0, 1, 1/2\}$. The meaning of φ is defined inductively as in Defn. A.2.1.*

We say that S and Z **potentially satisfy** *φ, denoted by $S, Z \models_3 \varphi$, if $[\![\varphi]\!]_3^S(Z) = 1/2$ or $[\![\varphi]\!]_3^S(Z) = 1$. We write $S \models_3 \varphi$ if for every Z we have $S, Z \models_3 \varphi$.*

Appendix B

Additional Proofs

B.1 Proofs for Chapter 4

For proving Theorem 4.4.5 we need a few additional definitions.

We first have to formally define $rep(\pi)$, and also introduce an intermediate assignment Z which will be used to record values of intermediate assignments through evaluation.

Definition B.1.1 (Trace representation) *Given a trace π, we define $rep(\pi) = \langle U_{rep(\pi)}, \iota_{rep(\pi)} \rangle$ to be the representation of π as a first-order logical structure, where:*

- *for every world π_i in π, there exists a world individual $w_i \in U_{rep(\pi)}$ s.t. $\iota_{rep(\pi)}(world)(w_i) = 1$.*
- *for every individual u in the universe U_{π_i} of a world π_i in the trace, there exists a corresponding non-world individual $\tilde{u} \in U_{rep(\pi)}$, s.t. $\iota_{rep(\pi)}(world)(\tilde{u}) = 0$.*
- *for every two successive worlds π_i and π_{i+1} in π, having corresponding world individuals $w_i, w_{i+1} \in U_{rep(\pi)}$, $\iota_{rep(\pi)}(succ)(w_i, w_{i+1}) = 1$.*
- *for the first world of the trace π_0 in π, having a corresponding world individual $w_0 \in U_{rep(\pi)}$, $\iota_{rep(\pi)}(initialWorld)(w_0) = 1$.*
- *for every world π_i with a corresponding world individual w_i, and for every individual $u \in U_{\pi_i}$ with a corresponding individual $\tilde{u} \in U_{rep(\pi)}$, $\iota_{rep(\pi)}(exists)(\tilde{u}, w_i) = 1$, and for every other world $\pi_j, j \neq i$ with a corresponding world individual w_j, $\iota_{rep(\pi)}(exists)(\tilde{u}, w_j) = 0$.*
- *for every two consecutive worlds π_i, π_{i+1} in π, and for every two individuals $u_i \in U_{\pi_i}$ and $u_{i+1} \in U_{\pi_{i+1}}$ with corresponding individuals $\tilde{u}_i, \tilde{u}_{i+1} \in U_{rep(\pi)}$, $\iota_{rep(\pi)}(evolution)(\tilde{u}_i, \tilde{u}_{i+1}) = 1$ iff $e_{\pi_i}(u_i) = u_{i+1}$.*
- *for every world π_i and an individual $u \in \pi_i$ with a corresponding individual $\tilde{u} \in U_{rep(\pi)}$, $\iota_{rep(\pi)}(isNew)(\tilde{u}) = 1$ iff $u \in A_{\pi_i}$.*

- *for every world* π_i *and an individual* $u \in \pi_i$ *with a corresponding individual* $\tilde{u} \in U_{rep(\pi)}$, $\iota_{rep(\pi)}(isFreed)(\tilde{u}) = 1$ *iff* $u \in D_{\pi_i}$.

We augment the notion of assignment as follows: an assignment Z assigns individuals from the universe to logical variables, and assigns a world of the trace to the designated logical variable w.

Definition B.1.2 *Given a trace* π*, an ETL formula* φ*, an assignment* Z*, and a world of the trace* π_i *for some* i*, we say that* $\pi, Z \models_t [\varphi]_w$ *when:*

- $Z(w) = \pi_i$
- $\pi^i, Z \models \varphi$

That is, when the suffix of π *starting from the world assigned to* w *satisfies the property.*

Lemma B.1.3 (Prefix Redundancy) *Given a trace* π *and an assignment* Z *assigning a world* π_i *to* w*,*

$$\pi, Z \models_t [\varphi]_w \iff \pi^i, Z \models_t [\varphi]_w$$

Proof:

$\pi, Z \models_t [\varphi]_w \iff$ (by Definition B.1.2)
$\pi^i, Z \models \varphi \iff$ (by Definition B.1.2)
$\pi^i, Z \models_t [\varphi]_w$

Definition B.1.4 *We define an additional operation on traces,* $\triangleright_w(\pi)$ *that takes a trace and a logical variable assigned by* Z *to a world in the trace. The operation returns the suffix of the trace starting at the given world.*

The initial assignment Z *assigns* w *to the first world of the trace.*

Proof:[Theorem 4.4.5]

We need to prove that for every closed ETL formula φ and a trace π, $\pi \models \varphi$ if and only if $rep(\pi) \models (\varphi)^\dagger$, where $rep(\pi)$ is the first-order representation of π. We will now show that:

$$\pi, Z \models_t [\varphi]_w \text{ if and only if } rep(\triangleright_w(\pi)), Z \models (\varphi)^{\dagger w}$$

0,1 trivially holds.

$\mathbf{p(v_1, \ldots, v_k)}$

$rep(\triangleright_w(\pi)), Z \models (p(v_1, \ldots, v_k))^{\dagger w} \iff$ (Definition 4.8.1)
$rep(\triangleright_w(\pi)), Z \models p(v_1, \ldots, v_k) \iff$ (Definition A.2.1)
$\iota_{head(\triangleright_w(\pi))}(p)(Z(v_1), \ldots, Z(v_k)) = 1 \iff$ (Definition 5.2.10)
$\pi^i, Z \models p(v_1, \ldots, v_k)$ where $Z(w) = \pi_i \iff$ (Definition B.1.2)
$\pi, Z \models_t [p(v_1, \ldots, v_k)]_w$

$\varphi \wedge \psi$

$rep(\triangleright_w(\pi)), Z \models (\varphi \wedge \psi)^{\dagger w} \iff$ (Definition 4.8.1)
$rep(\triangleright_w(\pi)), Z \models (\varphi)^{\dagger w}$ and $rep(\triangleright_w(\pi)), Z \models (\psi)^{\dagger w} \iff$ inductive assumption
$\pi, Z \models_t [\varphi]_w$ and $\pi, Z \models_t [\psi]_w \iff$ (Definition 5.2.10)
$\pi, Z \models_t [\varphi \wedge \psi]_w$

$\varphi \vee \psi$

$rep(\triangleright_w(\pi)), Z \models (\varphi \vee \psi)^{\dagger w} \iff$ (Definition 4.8.1)
$rep(\triangleright_w(\pi)), Z \models (\varphi)^{\dagger w}$ or $rep(\triangleright_w(\pi)), Z \models (\psi)^{\dagger w} \iff$ inductive assumption
$\pi, Z \models_t [\varphi]_w$ or $\pi, Z \models_t [\psi]_w \iff$ (Definition 5.2.10)
$\pi, Z \models_t [\varphi \vee \psi]_w$

$\exists \mathbf{v}.\varphi(\mathbf{v})$

$rep(\triangleright_w(\pi)), Z \models (\exists v.\varphi(v))^{\dagger w} \iff$ (Definition 4.8.1)
$rep(\triangleright_w(\pi)), Z \models \exists v.exists(w, v) \wedge (\varphi(v))^{\dagger w} \iff$
(assume only v is free without loss of generality)
exists $u \in U_{head(\triangleright_w(\pi))}$ s.t. $rep(\triangleright_w(\pi)), Z[v \mapsto u] \models (\varphi(v))^{\dagger w}$
exists $u \in U_{head(\triangleright_w(\pi))}$ s.t. $\pi, Z[v \mapsto u] \models_t [\varphi(v)]_w \iff$ (Definition 5.2.10)
$\pi, Z \models_t [\exists v.\varphi(v)]_w$

$(\mathbf{TC}\ \mathbf{v_1}, \mathbf{v_2} \colon \varphi)(\mathbf{v_3}, \mathbf{v_4})$

$rep(\triangleright_w(\pi)), Z \models ((TC\ v_1, v_2 \colon \varphi)(v_3, v_4))^{\dagger w} \iff$ (Definition 4.8.1)
$rep(\triangleright_w(\pi)), Z \models (TC\ v_1, v_2 \colon (\varphi)^{\dagger w} \wedge exists(w, v_1) \wedge exists(w, v_2))(v_3, v_4) \iff$
exists $u_1, \ldots, u_k \in U_T$. s.t. $Z(v_3) = u_1 \wedge Z(v_4) = u_k$
and for all $1 \leq i \leq k$.
$rep(\triangleright_w(\pi)), Z[v_1 \mapsto u_i, v_2 \mapsto u_{i+1}] \models (\varphi)^{\dagger w} \wedge exists(w, v_1) \wedge exists(w, v_2) \iff$
exists $u_1, \ldots, u_k \in U_{head(\triangleright_w(\pi))}$. s.t. $Z(v_3) = u_1 \wedge Z(v_4) = u_k$
and for all $1 \leq i \leq k. rep(\triangleright_w(\pi)), Z[v_1 \mapsto u_i, v_2 \mapsto u_{i+1}] \models (\varphi)^{\dagger w} \iff$
$\pi, Z \models_t [(TC\ v_1, v_2 \colon \varphi)(v_3, v_4)]_w$

$\bigcirc\varphi(\mathbf{x_1}, \ldots, \mathbf{x_n})$

$rep(\triangleright_w(\pi)), Z \models (\bigcirc\varphi(x_1, \ldots, x_n))^{\dagger w} \iff$ (Lemma B.1.5)
$rep(\triangleright_{w'}(\pi)), Z' \models (\varphi(x_1, \ldots, x_n))^{\dagger w'}$
where $succ(w, w')$, Z' is an evolution of $Z \iff$
$\pi, Z' \models_t [\varphi]_{w'} \iff$ chop worlds before w', Lemma B.1.3
$tail(\pi), Z' \models_t [\varphi]_{w'} \iff$ (Definition 5.2.10)
$\pi, Z \models_t [\bigcirc\varphi]_w$

$\varphi(\mathbf{x_1},\ldots,\mathbf{x_n})\mathcal{U}\psi(\mathbf{y_1},\ldots,\mathbf{y_n})$

$rep(\triangleright_w(\pi)), Z \models (\varphi(x_1,\ldots,x_n)\mathcal{U}\psi(y_1,\ldots,y_n))^{\dagger w} \iff$ (Definition 4.8.1)

$rep(\triangleright_w(\pi)), Z \models \exists w'\text{:}\, world.\exists y'_1,\ldots,y'_k.succ^*(w,w') \wedge (\psi(y'_1,\ldots,y'_k))^{\dagger w'}$
$\wedge \bigwedge_{1\leq i\leq k} evolution^*(y_i,y'_i) \wedge \forall\tilde{w}\text{:}\, world.\exists x'_1,\ldots,x'_n.(succ^*(w,\tilde{w})$
$\wedge succ^*(\tilde{w},w') \rightarrow (\varphi(x'_1,\ldots,x'_n))^{\dagger\tilde{w}} \wedge \bigwedge_{1\leq j\leq n} evolution^*(x_j,x'_j)) \iff$

exists $w', rep(\triangleright_w(\pi)), Z \models succ^*(w,w')$ and
exists $Z', rep(\triangleright_w(\pi)), Z' \models (\psi(y_1,\ldots,y_k))^{\dagger w'}$ and
for all $\tilde{w}, rep(\triangleright_w(\pi)), Z \models (succ^*(w,\tilde{w}) \wedge succ^*(\tilde{w},w')$ implies
exists $\tilde{Z}, rep(\triangleright_w(\pi)), Z \models (\varphi(x_1,\ldots,x_n))^{\dagger\tilde{w}} \iff$ (Lemma B.1.6)

exists $w', k \geq 0, s.t. Z(w') = \pi_k$ and
exists $Z', rep(\triangleright_w(\pi)), Z' \models (\psi(y_1,\ldots,y_k))^{\dagger w'}$ and
for all $\tilde{w}, rep(\triangleright_w(\pi)), Z \models (succ^*(w,\tilde{w}) \wedge succ^*(\tilde{w},w')$ implies
exists $\tilde{Z}, rep(\triangleright_w(\pi)), Z \models (\varphi(x_1,\ldots,x_n))^{\dagger\tilde{w}} \iff$

exists $w', k \geq 0, s.t. Z(w') = \pi_k$ and
exists $Z', rep(\triangleright_w(\pi)), Z' \models (\psi(y_1,\ldots,y_k))^{\dagger w'}$ and
for all $\tilde{w}, Z(\tilde{w}) = \pi_i, 1 \leq i \leq k$,
there exists $\tilde{Z}, rep(\triangleright_w(\pi)), Z \models (\varphi(x_1,\ldots,x_n))^{\dagger\tilde{w}} \iff$

exists $w', k \geq 0, s.t. Z(w') = \pi_k$ and
exists $Z', rep(\triangleright_{w'}(\pi)), Z' \models (\psi(y_1,\ldots,y_k))^{\dagger w'}$ and
for all $\tilde{w}, Z(\tilde{w}) = \pi_i, 1 \leq i \leq k$,
there exists $\tilde{Z}, rep(\triangleright_{\tilde{w}}(\pi)), Z \models (\varphi(x_1,\ldots,x_n))^{\dagger\tilde{w}} \iff$ (ind.)

exists $w', k \geq 0, s.t. Z(w') = \pi_k$ and
exists $Z', \pi, Z' \models_t [\psi(y_1,\ldots,y_k)]_{w'}$ and
for all $\tilde{w}, Z(\tilde{w}) = \pi_i, 1 \leq i \leq k$,
there exists $\tilde{Z}, \pi, \tilde{Z} \models_t [\varphi(x_1,\ldots,x_n)]_{\tilde{w}} \iff$ (Lemma B.1.3)

exists $w', k \geq 0, s.t. Z(w') = \pi_k$ and
exists $Z', \pi^k, Z' \models_t [\psi(y_1,\ldots,y_k)]_{w'}$ and
for all $\tilde{w}, Z(\tilde{w}) = \pi_i, 1 \leq i \leq k$,
there exists $\tilde{Z}, \pi^i, \tilde{Z} \models_t [\varphi(x_1,\ldots,x_n)]_{\tilde{w}} \iff$ (Def. B.1.2 + Definition 5.2.10)

$\pi, Z \models_t [\varphi(x_1,\ldots,x_n)\mathcal{U}\psi(y_1,\ldots,y_n)]_w$

Lemma B.1.5

$$rep(\triangleright_w(\pi)), Z \models (\bigcirc\varphi(x_1,\ldots,x_n))^{\dagger w} \iff rep(\triangleright_{w'}(\pi)), Z' \models (\varphi(x_1,\ldots,x_n))^{\dagger w'}$$

where $succ(w,w')$ and Z' is the evolution of Z.

Proof:

$rep(\triangleright_w(\pi)), Z \models (\bigcirc\varphi(x_1,\ldots,x_n))^{\dagger w} \iff$

$rep(\triangleright_w(\pi)), Z \models \exists w'\!:world.\exists x'_1,\ldots,x'_n.succ(w,w') \wedge (\varphi(x'_1,\ldots,x'_n))^{\dagger w'} \wedge$

$\bigwedge_{1\leq j\leq n} evolution(x_j,x'_j) \wedge exists(x'_j,w') \iff$

(when $succ(w,w')$)

$rep(\triangleright_{w'}(\pi)), Z \models \exists x'_1,\ldots,x'_n.(\varphi(x'_1,\ldots,x'_n))^{\dagger w'} \wedge \bigwedge_{1\leq j\leq n} evolution(x_j,x'_j) \wedge exists(x'_j,w') \iff$

(when Z' is the evolution of Z)

$rep(\triangleright_{w'}(\pi)), Z' \models (\varphi(x_1,\ldots,x_n))^{\dagger w'}$

Lemma B.1.6 *Given a trace π, an assignment Z assigning a world individual to the logical variable w, for any w',*

$$rep(\triangleright_w(\pi)), Z \models succ^*(w,w') \iff \textit{there exists } k\geq 0, Z(w') = \pi_k$$

Proof: Trivial from definition of TC operator. This lemma is provided to emphasize that $succ^*(w,w')$ corresponds to the existence of a successor within a *finite future*.

B.1.1 Embedding Theorem

Proof:[Embedding Theorem for Infinite Configurations, Theorem 4.5.6] This theorem generalizes the embedding theorem of [91] for the infinite case. The proof is identical to the proof given for the original embedding theorem since the same arguments hold for the infinite case. The proof is by structural induction on φ:

Basis: For atomic formula $p(v_1,v_2,\ldots,v_k)$, $u_1,u_2,\ldots,u_k \in U^S$, and $Z = [v_1\mapsto u_1, v_2\mapsto u_2,\ldots,v_k\mapsto u_k]$ we have

$$\begin{aligned}
&[\![p(v_1,v_2,\ldots,v_k)]\!]^S_3(Z)\\
&\quad = \iota^S(p)(u_1,u_2,\ldots,u_k) && \text{(Definition A.3.1)}\\
&\quad \sqsubseteq \iota^{S'}(p)(f(u_1),f(u_2),\ldots,f(u_k)) && \text{(Definition 4.5.3)}\\
&\quad = [\![p(v_1,v_2,\ldots,v_k)]\!]^{S'}_3(f\circ Z) && \text{(Definition A.3.1)}
\end{aligned}$$

Also, for $l\in\{0,1,1/2\}$, we have:

$$\begin{aligned}
&[\![\boldsymbol{l}]\!]^S_3(Z)\\
&\quad = l && \text{(Definition A.3.1)}\\
&\quad \sqsubseteq l && \text{(Definition 4.5.2)}\\
&\quad = [\![\boldsymbol{l}]\!]^{S'}_3(f\circ Z) && \text{(Definition A.3.1)}
\end{aligned}$$

Induction step: Suppose φ is a formula with free variables $v_1,v_2,\ldots v_k$. Let Z be a complete assignment for φ. If $[\![\varphi]\!]^{S'}_3(Z) = 1/2$, then the theorem holds trivially. Therefore assume that $[\![\varphi]\!]^{S'}_3(f\circ Z)\in\{0,1\}$. We distinguish between the following cases:

Logical-and $\varphi \equiv \varphi_1 \wedge \varphi_2$. The proof splits into the following subcases:

Case 1: $[\![\varphi_1 \wedge \varphi_2]\!]_3^{S'}(f \circ Z) = 0$.
In this case, either $[\![\varphi_1]\!]_3^{S'}(f \circ Z) = 0$ or $[\![\varphi_2]\!]_3^{S'}(f \circ Z) = 0$. Without loss of generality assume that $[\![\varphi_1]\!]_3^{S'}(f \circ Z) = 0$. Then, by the induction hypothesis for φ_1, we conclude that $[\![\varphi_1]\!]_3^{S}(Z) = 0$. Therefore, by Definition A.3.1, $[\![\varphi_1 \wedge \varphi_2]\!]_3^{S}(Z) = 0$.

Case 2: $[\![\varphi_1 \wedge \varphi_2]\!]_3^{S'}(f \circ Z) = 1$.
In this case, both $[\![\varphi_1]\!]_3^{S'}(f \circ Z) = 1$ and $[\![\varphi_2]\!]_3^{S'}(f \circ Z) = 1$. Then, by the induction hypothesis for φ_1 and φ_2, we conclude that $[\![\varphi_1]\!]_3^{S}(Z) = 1$ and
$[\![\varphi_2]\!]_3^{S}(Z) = 1$. Therefore, by Definition A.3.1, $[\![\varphi_1 \wedge \varphi_2]\!]_3^{S}(Z) = 1$.

Logical-negation $\varphi \equiv \neg\varphi_1$. The proof splits into the following subcases:

Case 1: $[\![\neg\varphi_1]\!]_3^{S'}(f \circ Z) = 0$.
In this case, $[\![\varphi_1]\!]_3^{S'}(f \circ Z) = 1$.
Then, by the induction hypothesis for φ_1, we conclude that $[\![\varphi_1]\!]_3^{S}(Z) = 1$.
Therefore, by Definition A.3.1, $[\![\neg\varphi_1]\!]_3^{S}(Z) = 0$.

Case 2: $[\![\neg\varphi_1]\!]_3^{S'}(f \circ Z) = 1$.
In this case, $[\![\varphi_1]\!]_3^{S'}(f \circ Z) = 0$.
Then, by the induction hypothesis for φ_1, we conclude that $[\![\varphi_1]\!]_3^{S}(Z) = 0$.
Therefore, by Definition A.3.1, $[\![\neg\varphi_1]\!]_3^{S}(Z) = 1$.

Existential-Quantification $\varphi \equiv \exists v_0 : \varphi_1$. The proof splits into the following subcases:

Case 1: $[\![\exists v_1 : \varphi_1]\!]_3^{S'}(f \circ Z) = 0$.
In this case, for all $u \in U^S$, $[\![\varphi_1]\!]_3^{S'}((f \circ Z)[v_1 \mapsto f(u)]) = 0$. Then, by the induction hypothesis for φ_1, we conclude that for all $u \in U^S$ $[\![\varphi_1]\!]_3^{S}(Z[v_1 \mapsto u]) = 0$. Therefore, by Definition A.3.1, $[\![\exists v_1 : \varphi_1]\!]_3^{S}(Z) = 0$.

Case 2:$[\![\exists v_1 : \varphi_1]\!]_3^{S'}(f \circ Z) = 1$.
In this case, there exists a $u' \in U^{S'}$ such that $[\![\varphi_1]\!]_3^{S'}((f \circ Z)[v_1 \mapsto u']) = 1$. Because f is surjective, there exists a $u \in U^S$ such that $f(u) = u'$ and $[\![\varphi_1]\!]_3^{S'}((f \circ Z)[v_1 \mapsto f(u)]) = 1$. Then, by the induction hypothesis for φ_1, we conclude that $[\![\varphi_1]\!]_3^{S}(Z[v_1 \mapsto u]) = 1$. Therefore, by Definition A.3.1, $[\![\exists v_1 : \varphi_1]\!]_3^{S}(Z) = 1$.

Transitive Closure $\varphi \equiv (TC\ v_1, v_2 \colon \varphi_1)(v_3, v_4)$. The proof splits into the following subcases:

Case 1: $[\![(TC\ v_1, v_2 \colon \varphi_1)(v_3, v_4)]\!]_3^{S'}(f \circ Z) = 1$.
By Definition A.3.1, there exist $u'_1, u'_2, \ldots, u'_{n+1} \in U^{S'}$ such that for all $1 \leq i \leq n$, $[\![\varphi_1]\!]_3^{S'}((f \circ Z)[v_1 \mapsto u'_i, v_2 \mapsto u'_{i+1}]) = 1$, $(f \circ Z)(v_3) = u'_1$, and $(f \circ Z)(v_4) = u'_{n+1}$. Because f is surjective, there exist $u_1, u_2, \ldots, u_{n+1} \in U^S$ such that for all $1 \leq i \leq n+1$, $f(u_i) =$

u'_i. Therefore, $Z(v_3) = u_1$, $Z(v_4) = u_{n+1}$, and by the induction hypothesis, for all $1 \leq i \leq n$, $[\![\varphi_1]\!]^S_3(Z[v_1 \mapsto u_i, v_2 \mapsto u_{i+1}]) = 1$. Hence, by Definition A.3.1, $[\![(TC\ v_1, v_2 \colon \varphi_1)(v_3, v_4)]\!]^S_3(Z) = 1$.

Case 2: $[\![(TC\ v_1, v_2 \colon \varphi_1)(v_3, v_4)]\!]^{S'}_3(f \circ Z) = 0$.
We need to show that $[\![(TC\ v_1, v_2 \colon \varphi_1)(v_3, v_4)]\!]^S_3(Z) = 0$. Assume on the contrary that

$$[\![(TC\ v_1, v_2 \colon \varphi_1)(v_3, v_4)]\!]^{S'}_3(f \circ Z) = 0$$

, but $[\![(TC\ v_1, v_2 \colon \varphi_1)(v_3, v_4)]\!]^S_3(Z) \neq 0$. Because $[\![(TC\ v_1, v_2 \colon \varphi_1)(v_3, v_4)]\!]^S_3(Z) \neq 0$, by Definition A.3.1 there exist $u_1, u_2, \ldots, u_{n+1} \in U^S$ such that $Z(v_3) = u_1$, $Z(v_4) = u_{n+1}$, and for all $1 \leq i \leq n$, $[\![\varphi_1]\!]^S_3(Z[v_1 \mapsto u_i, v_2 \mapsto u_{i+1}]) \neq 0$. Hence, by the induction hypothesis there exist $u'_1, u'_2, \ldots, u'_{n+1} \in U^{S'}$ such that $(f \circ Z)(v_3) = u'_1$, and $(f \circ Z)(v_4) = u'_{n+1}$ and for all $1 \leq i \leq n$, $[\![\varphi_1]\!]^{S'}_3((f \circ Z)[v_1 \mapsto u'_i, v_2 \mapsto u'_{i+1}]) \neq 0$. Therefore, by Definition A.3.1, $[\![(TC\ v_1, v_2 \colon \varphi_1)(v_3, v_4)]\!]^{S'}_3(f \circ Z) \neq 0$, which is a contradiction.

B.2 Proofs for Chapter 5

Proof:[Theorem 5.2.13] We will show that given a BDETL formula φ, and a program P, $P \models \varphi \implies P \models_E \{\varphi\}$, by proving that for every program trace π, $\pi, Z \models \varphi \implies c, Z \models_E \{\varphi\}$ where $c = head(\pi)$

$\varphi = 1$

$$\pi, Z \models 1 \implies head(\pi), Z \models_E \{1\}$$

$\varphi = 0$

$$\pi, Z \models 0 \implies head(\pi), Z \models_E \{0\}$$

$\varphi = p(v_1, \ldots, v_k)$

$$\begin{aligned} &\pi, Z \models p(v_1, \ldots, v_k) \iff \\ &\iota_{head(\pi)}(p)(Z(v_1), \ldots, Z(V_k)) = 1 \iff \\ &head(\pi), Z \models_E \{p(v_1, \ldots, v_k)\} \end{aligned}$$

$\varphi = \neg\varphi_a$ **where** φ_a **is an atomic formula**

$$\begin{aligned} &\pi, Z \models \neg\varphi_a \iff \\ &\text{not } \pi, Z \models \varphi_a \iff \\ &head(\pi), Z \models_E \{\neg\varphi_a\} \end{aligned}$$

$\varphi = \varphi_1 \vee \varphi_2$

$\pi, Z \models \varphi_1 \vee \varphi_2 \iff$
$\pi, Z \models \varphi_1$ or $\pi, Z \models \varphi_2 \implies$
$head(\pi), Z \models_E \{\varphi_1\}$ or $head(\pi), Z \models_E \{\varphi_2\} \iff$
$head(\pi), Z \models_E \{\varphi_1 \vee \varphi_2\}$

$\varphi = \varphi_1 \wedge \varphi_2$

$\pi, Z \models \varphi_1 \wedge \varphi_2 \iff$
$\pi, Z \models \varphi_1$ and $\pi, Z \models \varphi_2 \implies$
$head(\pi), Z \models_E \{\varphi_1\}$ and $head(\pi), Z \models_E \{\varphi_2\} \iff$
$head(\pi), Z \models_E \{\varphi_1 \wedge \varphi_2\}$

$\varphi = \exists v.\varphi_1$

$\pi, Z \models \exists v.\varphi_1(v) \iff$
exists $u \in U_{head(\pi)}$ s.t. $\pi, Z[v \mapsto u] \models \varphi_1(v) \implies$
exists $u \in U_{head(\pi)}$ s.t. $head(\pi), Z[v \mapsto u] \models_E \{\varphi_1(v)\} \iff$
$head(\pi), Z[v \mapsto u] \models_E \{\exists v.\varphi_1(v)\}$

$\varphi = \bigcirc\varphi_1$

$\pi, Z \models \bigcirc\varphi_1 \iff$
$tail(\pi), Z \models \varphi_1 \implies$
$c', Z \models_E \varphi_1$ where $c' = head(tail(\pi)) \implies$
there exists a successor c' of $head(\pi)$, s.t. $c', Z \models_E \varphi_1 \implies head(\pi), Z \models_E \{\bigcirc\varphi_1\}$

$\varphi = \varphi_1 \mathcal{U} \varphi_2$

$\pi, Z \models \varphi_1 \mathcal{U} \varphi_2 \iff$
there exists $k \geq 0, \pi^k, Z \models \varphi_2$ and for all $0 \leq j \leq k, \pi^j, Z \models \varphi_1$
we will show that this implies $head(\pi), Z \models_E \{\varphi_1 \mathcal{U} \varphi_2\}$
by induction on k
base: $k = 0$
$\pi^0, Z \models \varphi_2 \implies head(\pi), Z \models_E \{\varphi_2\} \implies$
$head(\pi), Z \models_E \{\varphi_1 \mathcal{U} \varphi_2\}$
step: $k > 0$
there exists $k > 0, \pi^k, Z \models \varphi_2$ and for all $0 \leq j \leq k, \pi^j, Z \models \varphi_1$
$\implies \pi, Z \models \varphi_1 \implies head(\pi), Z \models_E \{\varphi_1\}$
now consider $\pi' = tail(\pi)$
there exists $k' = k - 1 \geq 0, \pi'^{k'}, Z \models \varphi_2$ and for all $0 \leq j \leq k', \pi'^j, Z \models \varphi_1$

$\varphi = \varphi_1 \mathcal{W} \varphi_2$

$\pi, Z \models \varphi_1 \mathcal{W} \varphi_2 \iff$
there exists $k \geq 0, \pi^k, Z \models \varphi_2$ and for all $0 \leq j \leq k, \pi^j, Z \models \varphi_1$
or for all $j \geq 0, \pi^j, Z \models \varphi_1$
we will show that this implies $head(\pi), Z \models_E \{\varphi_1 \mathcal{W} \varphi_2\}$
case 1: there exists $k \geq 0, \pi^k, Z \models \varphi_2$
by induction on k
base: $k = 0$
$\pi^0, Z \models \varphi_2 \implies head(\pi), Z \models_E \{\varphi_2\} \implies$
$head(\pi), Z \models_E \{\varphi_1 \mathcal{W} \varphi_2\}$
step: $k > 0$
there exists $k > 0, \pi^k, Z \models \varphi_2$ and for all $0 \leq j \leq k, \pi^j, Z \models \varphi_1$
$\implies \pi, Z \models \varphi_1 \implies head(\pi), Z \models_E \{\varphi_1\}$
now consider $\pi' = tail(\pi)$
there exists $k' = k - 1 \geq 0, \pi'^{k'}, Z \models \varphi_2$ and for all $0 \leq j \leq k', \pi'^j, Z \models \varphi_1$
case 2: no $k \geq 0$ exists s.t. $\pi^k, Z \models \varphi_2$
for all $j \geq 0, \pi^j, Z \models \varphi_1 \implies$
for all $j \geq 1, \pi^j, Z \models \varphi_1$ and $\pi^0, Z \models \varphi_1 \implies$
$\pi^1, Z \models \varphi_1 \mathcal{W} \varphi_2$ and $\pi^0, Z \models \varphi_1 \implies$
$head(tail(\pi)), Z \models_E \{\varphi_1 \mathcal{W} \varphi_2\}$ and $head(\pi), Z \models_E \{\varphi_1\} \implies$
$head(\pi), Z \models_E \{\varphi_1 \mathcal{W} \varphi_2\}$

Proof:[Theorem 5.3.5] We show that the abstract state-based semantics is an abstract interpretation of the concrete state-based semantics of Definition 5.2.12. We will show that

$$c \models_E F \implies \mathit{blur}(c), \models^\sharp_E F$$

In order to show that, we establish a Galois connection between concrete and abstract configurations.

$$\beta(c) = \mathit{blur}(c)$$
$$\alpha(\mathcal{C}) = \bigcup_{c \in \mathcal{C}} \mathit{blur}(c)$$
$$\gamma(\mathcal{A}) = \{c \mid \mathit{blur}(c) \in \mathcal{A}\}$$

Appendix C

ETL Supplements

C.1 Additional Properties for Mark And Sweep

Fig. C.1 shows the code for the sweep phase of the mark and sweep collector. The properties of interest for the sweep phase are formulated as the following ETL formulae.

$$\text{(S1)} \quad \forall^{\epsilon} v. \neg marked(v) \rightarrow \Diamond \oslash (v)$$

$$\text{(S2)} \quad \Box \forall^{\epsilon} v. \oslash (v) \rightarrow \neg marked(v)$$

The progress measure required to verify these properties is:

$$\varphi_{\downarrow} = \exists^{\epsilon} v. pending(v) \wedge \neg pending'(v))$$

$$\varphi_{\uparrow} = \exists^{\epsilon} v. \neg pending(v) \wedge pending'(v))$$

C.2 Additional ETL Properties

Table C.1 presents a list of simple programs and program properties specified via ETL. The program `Mutex` is a program that dynamically allocates an unbounded number of threads and lets them compete for a critical section protected by a single lock [115]. The program `Two lock queue` is an implementation of a concurrent shared queue protected by two locks [115]. The program `DelAll` is a simple sequential program that deletes all the elements of a given linked list. The properties $free$ and $assign - null$ could be used for performing compile-time garbage collection, as done in [95]. The `Web Server` program is a simple implementation of a web-server as used in [119].

C.3 ETL with Past Operators

This appendix completes the partial (but less cumbersome) definitions of ETL trace-based and state-based semantics given in Chapter 5.

Program	**Property**	**S/L**	**UB**	**Comments**
Mutex	$\forall t.\ \Diamond\, at[l_{crit}](t)$	L	T	absence of starvation
	$\Diamond\, \exists t.at[l_{crit}](t)$	L	T	progress
	$\exists t.\ \Diamond\, at[l_{crit}](t)$	L	T	specific progress
	$\Box\, \forall t_1, t_2.\neg(at[l_{crit}](t_1) \wedge at[l_{crit}](t))$	S	T	mutex
Two lock queue	$\Box\, \exists v.tail(v) \rightarrow rf[head, next](v)$	S	O/T	queue connected
	$\Box\, \exists t_i, v, u.at[lp_6](t_i) \wedge rv[x_i](t_i, v)$ $\wedge\, rf[head, next](v) \wedge tail(u) \rightarrow rval[next](u, v)$	S	O/T	insert after last
	$\Box\, \exists t_d, v.at[lt_9](t_d) \wedge rv[x_d](t_d, v)$ $\wedge\, rf[head, next](v) \rightarrow head(v)$	S	O/T	delete first
	$\Box\ \nexists v, u.head(v) \wedge rv[next](u, v)$	S	O/T	head is first
	$\Box\, \forall t.\neg at[li_4](t) \rightarrow \exists v\colon tail(v)$	S	O/T	tail exists
	$\Box\, \forall t.at[lp_2](t) \rightarrow \Diamond\, at[lp_5](t)$	L	O/T	producer liveness
	$\Box\, \forall t.at[lt_1](t) \rightarrow \Diamond\, at[lt_8](t)$	L	O/T	consumer liveness
DelAll	$\forall v.\ \Diamond\, x(v)$	L	O	all items eventually traversed by x
	$\Diamond\, at[l_{exit}]()$	L	O	termination
various	$\Box(\forall v.at[pt]() \wedge y(v) \rightarrow \bigcirc\, \Box\, \neg use(v))$	S	O	$free\langle pt, y\rangle$
various	$\Box(\forall v.at[pt]() \wedge y(v) \rightarrow \bigcirc \neg use_n(v)\, \mathcal{W} def_n(v))$	S	O	$asgn - null\langle pt, y\rangle$
Web server	$\Box\, \forall t_1, t_2\colon thread.(t_1 \neq t_2)$ $\rightarrow \neg(at[lw_c](t_1) \wedge at[lw_c](t_2))$	S	O/T	mutex over the shared resource
	$\Box\, \forall t\colon thread.at[lw_1](t) \rightarrow \Diamond\, at[lw_c](t)$	L	O/T	absence of starvation for worker threads
	$\Box(\forall t\colon thread.\neg \odot t) \vee$ $(\forall t\colon thread.\neg \odot t)\, \mathcal{U}\, (\exists v\colon request.\ \odot v)$	S	O/T	a thread only created when request received
	$\Box\, \exists v\colon request.\ \odot v \rightarrow \Diamond\, \exists t\colon thread.\ \odot t$	L	O/T	each request followed by thread creation
	$\Box\, \forall t_1, t_2\colon thread.(t_1 \neq t_2)$ $\rightarrow \neg(at[ls_2](t_1) \wedge at[la_3](t_2))$	S	O/T	mutex of listener and scheduler over sched. queue
	$\Box\, \forall t\colon thread.\ \odot t$ $\rightarrow \Diamond\, \exists q\colon queue.rval[head.next^*](q, t)$	L	O/T	each created thread is eventually inserted into the sched. queue
	$\Box\, \forall t\colon thread.at[lw_1](t)$ $\rightarrow \neg\exists q\colon queue.rval[head.next^*](q, t)$	L	O/T	each scheduled worker thread removed from sched. queue
	$\exists q\colon queue.\ \Box\, \forall t\colon thread.$ $(rval[head.next^*](q, t))$ $\rightarrow \Diamond\, \neg(rval[head.next^*](q, t))$	L	O/T	each worker thread waiting in queue eventually leaves queue

Table C.1: Example programs and ETL specifications

```
   public Set sweep(Set marked, Element root) {
s1    Set pending = Heap.universe;
s2    Set collected = new HashSet();
s3    if (root != null) {
s4       while (!pending.isEmpty()) {
s5          Element x = (Element) pending.iterator().next();
s6          pending.remove(x);
s7          if (!marked.contains(x))
s8             collected.add(x);
s9          else
s10            marked.remove(x);
s11      }
s12   }
s13   return collected;
   }
```

Figure C.1: Java source for the sweep-phase procedure.

Definition C.3.1 (ETL Trace-based Semantics) *We define when an ETL formula φ is satisfied over a trace π starting at index i of the trace with an assignment Z (denoted by $\pi, i, Z \models \varphi$) as follows:*

- $\pi, i, Z \models \mathbf{1}$, *and not* $\pi, i \models \mathbf{0}Z$.
- $\pi, i, Z \models p(v_1, \ldots, v_k)$ *when* $\iota_{\pi_i}(p)(Z(v_1), \ldots, Z(v_k)) = 1$
- $\pi, i, Z \models \neg\varphi$ *when not* $\pi, i, Z \models \varphi$
- $\pi, i, Z \models \varphi \vee \psi$ *when* $\pi, i, Z \models \varphi$ *or* $\pi, i \models \psi Z$
- $\pi, i, Z \models \exists v.\varphi(v)$ *when there exists* $u \in U$ *s.t.* $\pi, i, Z[v \mapsto u] \models \varphi(v)$
- $\pi, i, Z \models (TC\ v_1, v_2 \colon \varphi)(v_3, v_4)$ *when there exists* $u_1, \ldots, u_{n+1} \in U$, *s.t.* $Z(v_3) = u_1, Z(v_4) = u_{n+1}$, *and for all* $1 \leq j \leq n, \pi, i, Z[v_1 \mapsto u_j, v_2 \mapsto u_{j+1}] \models \varphi$.
- $\pi, i, Z \models \bigcirc\varphi$ *when* $\pi, i+1, Z \models \varphi$.
- $\pi, i, Z \models \varphi \mathcal{U} \psi$ *when there exists* $k \geq i$, *s.t.,* $\pi, k, Z \models \psi$ *and for all* $i \leq j < k$, $\pi, j, Z \models \varphi$.
- $\pi, i, Z \models \varphi \mathcal{W} \psi$ *when there exists* $k \geq i$, *s.t.,* $\pi, k, Z \models \psi$ *and for all* $i \leq j < k$, $\pi, j, Z \models \varphi$, *or for all* $j \geq i$, $\pi, j, Z \models \varphi$.
- $\pi, i, Z \models \ominus\, \varphi$ *when* $\pi, i-1, Z \models \varphi$.
- $\pi, i, Z \models \varphi \mathcal{S} \psi$ *when there exists* $0 \leq k \leq i$, *s.t.,* $\pi, k, Z \models \psi$ *and for all* $k \leq j < i$, $\pi, j, Z \models \varphi$.

- $\pi, i, Z \models \varphi \mathcal{B} \psi$ *when there exists* $0 \leq k \leq i$*, s.t.,* $\pi, k, Z \models \psi$ *and for all* $k \leq j < i$*,* $\pi, j, Z \models \varphi$*, or for all* $0 \leq j < i$*,* $\pi, j, Z \models \varphi$*.*

We omit definitions for $\wedge$*,* $\forall$*, since they are defined similarly. We write* $\pi \models \varphi$ *when* $\pi, 0, Z \models \varphi$ *for every assignment* Z*. Given a program* P*, we say that* $P \models \varphi$ *when there exists a trace* π *of the program* P*, such that* $\pi \models \varphi$*.*

Definition C.3.2 (ETL Existential State-Based Semantics) *Given a set of BDETL formulae* F*, and a program* P*, we say that* F *is* ***existentially satisfied*** *from a configuration (state)* $C^\natural$ *with an assignment* Z *(denoted by* $C^\natural, Z \models_E F$*) when one of the following conditions holds:*

(A0) $F = \emptyset$

(A1) $F = F' \cup \{\mathbf{1}\}$ *and* $C^\natural, Z \models_E F'$*,*

(A2) $F = F' \cup \{p(v_1, \ldots, v_k)\}$ *and* $\iota_{C^\natural}(p)(Z(v_1), \ldots, Z(v_k))$=1*, and* $C^\natural, Z \models_E F'$

(A3) $F = F' \cup \{\neg\varphi\}$ *and not* $C^\natural, Z \models_E \{\varphi\}$*, and* $C^\natural, Z \models_E F'$

(A4) $F = F' \cup \{\varphi \vee \psi\}$ *and* $C^\natural, Z \models_E F' \cup \{\varphi\}$ *or* $C^\natural, Z \models_E F' \cup \{\psi\}$

(A5) $F = F' \cup \{\varphi \wedge \psi\}$ *and* $C^\natural, Z \models_E F' \cup \{\varphi, \psi\}$

(A6) $F = F' \cup \{\exists v.\varphi(v)\}$ *and there exists* $u \in U_{C^\natural}$ *s.t.* $C^\natural, Z[v \mapsto u] \models_E F' \cup \{\varphi(v)\}$

(A7) $F = F' \cup \{\bigcirc\varphi\}$ *and exists* $C^{\natural\prime}$*,* $C^\natural \Rightarrow C^{\natural\prime}$ *s.t.,* $C^{\natural\prime}, Z \models_E \{\varphi\}$ *and* $C^\natural, Z \models_E F'$*.*

(A8) $F = F' \cup \{\varphi \mathcal{U} \psi\}$ *and* $C^\natural, Z \models_E F' \cup \{\psi\}$ *or*
$C^\natural, Z \models_E F' \cup \{\varphi\}$ *and there exists* $C^{\natural\prime}$ *s.t.* $C^\natural \Rightarrow C^{\natural\prime}$ *and* $C^{\natural\prime}, Z \models_E \{\varphi \mathcal{U} \psi\}$*.*

(A9) $F = F' \cup \{\varphi \mathcal{W} \psi\}$ *and* $C^\natural, Z \models_E F' \cup \{\psi\}$ *or*
$C^\natural, Z \models_E F' \cup \{\varphi\}$ *and there exists* $C^{\natural\prime}$ *s.t.* $C^\natural \Rightarrow C^{\natural\prime}$ *and* $C^{\natural\prime}, Z \models_E \{\varphi \mathcal{W} \psi\}$*.*

(A10) $F = F' \cup \{\ominus \varphi\}$ *and exists* $C^{\natural\prime}$*,* $C^\natural \Leftarrow C^{\natural\prime}$ *s.t.,* $C^{\natural\prime}, Z \models_E \{\varphi\}$ *and* $C^\natural, Z \models_E F'$*.*

(A11) $F = F' \cup \{\varphi \mathcal{S} \psi\}$ *and* $C^\natural, Z \models_E F' \cup \{\psi\}$ *or*
$C^\natural, Z \models_E F' \cup \{\varphi\}$ *and there exists* $C^{\natural\prime}$ *s.t.* $C^\natural \Leftarrow C^{\natural\prime}$ *and* $C^{\natural\prime}, Z \models_E \{\varphi \mathcal{S} \psi\}$*.*

(A12) $F = F' \cup \{\varphi \mathcal{B} \psi\}$ *and* $C^\natural, Z \models_E F' \cup \{\psi\}$ *or*
$C^\natural, Z \models_E F' \cup \{\varphi\}$ *and there exists* $C^{\natural\prime}$ *s.t.* $C^\natural \Leftarrow C^{\natural\prime}$ *and* $C^{\natural\prime}, Z \models_E \{\varphi \mathcal{B} \psi\}$*.*

where $\Leftarrow$ *is the reverse transition relation of* $\Rightarrow$*.*

Index