Verifying Safety Properties of Concurrent Heap-Manipulating Programs

Eran Yahav
and
Mooly Sagiv

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

—Concurrent manipulation of linked-list based ADT preserves the ADT datatype invariant.
—The program does not perform inconsistent updates due to interference.
—The program does not reach a deadlock.
—The program does not produce run-time errors due to illegal thread interactions.

We also found bugs in erroneous programs violating such properties. A prototype of our framework has been implemented and applied to small, but interesting, example programs.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*symbolic execution*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*data types and structures; dynamic storage management*; E.1 [**Data**]: Data Structures—*graphs; lists; trees*; E.2 [**Data**]: Data Storage Representations—*composite structures; linked representations*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*assertions; invariants*

General Terms: Algorithms, Languages, Theory, Verification

Additional Key Words and Phrases: Abstract Interpretation, Verification, Concurrency, Shape-analysis, Safety Properties, Java

Author's address: E. Yahav, IBM T.J. Watson Research Center, Hawthorne, NY
M. Sagiv, School of Computer Science, Tel Aviv University, Tel Aviv, 69978, Israel.

A preliminary version of this work appeared in the 28th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages and in the SoftMC 2003 Workshop on Software Model Checking.
This research was supported by a grant from the Ministry of Science, Israel, and also partially supported by the Israeli Academy of Science.

1. INTRODUCTION

Modern languages such as Java and C# provide low-level concurrency-control constructs that enable the programmer to create complicated and powerful synchronization schemes. However, these languages provide no means of compile-time or run-time checking for the correctness of concurrent behavior. This makes concurrent programming in these languages quite error-prone (e.g., [Vermeulen 1997; Lea 1997; Goetz et al. 2006]).

The theme of this paper is to develop static analysis techniques for verifying safety properties by detecting program configurations that may violate them. 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).

1.1 Main Results

In this paper, we present a framework for verifying safety properties of concurrent heap-manipulating 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., [Lea 1997], 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 with sufficient precision to show that they maintain subtle properties of interest.

1.1.1 *A Parametric Framework for Verifying Safety Properties.* We provide a parametric framework for verifying safety properties of concurrent heap-manipulating programs (A preliminary version of the framework appeared in [Yahav 2001]). We use different instances of this framework (see Section 1.1.2) 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., [Knapp et al. 1998]) 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 [Cousot and Cousot 1977]. 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 is parametric in the following: (i) the definition of a configuration (Sections 3.1, 7.2.1); (ii) the (concrete) operational semantics (Section 3.3); (iii) the definitions of properties to be verified (Sections 3.2, 3.4, 7.2.2); (iv) the manner in which concrete configurations are abstracted (Section 4.3 and Section 7.3).

Our framework can be viewed as on-the-fly model checking [Clarke et al. 1999a] for verifying safety properties of programs. On-the-fly model checking does not require the

```
while(?) {
  Thread t = new Thread(new WorkerRunner());
  t.start();
}
```

Fig. 1. A program allocating a number of worker threads (unknown a priori) at a single allocation site, demonstrating loss of precision in the two-phased approach.

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 simple *state-space exploration* algorithm (see Fig. 5) 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.

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 ([Clarke et al. 1994; Clarke et al. 1999b]). This could be trivially encoded in our framework by using only nullary predicates (e.g., see [Manevich et al. 2005]). 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 [Emerson and Sistla 1993] (formulated for processes rather than threads) relies on explicit process names, and thus cannot handle dynamic allocation of processes.

ESP [Das et al. 2002] and SLAM [Ball et al. 2001] 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. In contrast, our framework handles concurrent programs, and applies integrated verification and pointer analysis which is more precise.

For example, trying to verify correct thread usage for the program of Fig. 1 using a two-phased approach based on points-to analysis would yield a false-alarm — reporting that a thread may be started more than once (`IllegalThreadStateException`). The reason for this loss of precision is that all threads allocated at the same allocation site are represented using a single "abstract object". As a result, the start operation may appear as being possibly applied multiple times to a single thread object. The same program would be successfully verified using the integrated approach in which the state of the thread refines the heap abstraction, and makes observable the fact that the start operations are applied to different threads.

The reader may find our comparison to related works somewhat unfair in that we only compare the relative precision of the approaches, and not their scalability. However, in a practical sense, both our approach and the two-phased approaches are limited. Our approach will not yet scale (as is) to programs of realistic size, and the two-phased approaches will not be precise enough to verify many properties of interest.

Nevertheless, we believe that the scalability of our approach could be improved without loss of precision by considering a more limited setting and by using techniques such as dynamic partial-order reduction (e.g., [Gueta et al. 2006]), and staging (e.g., [Fink et al. 2006]). For example, Gotsman et al. [2007] present an analysis that is potentially more scalable by considering a more limited setting which requires knowledge about locks (and cannot handle fine-grained synchronization).

Recently, it was shown that thread-quantification [Berdine et al. 2008] and separation [Manevich et al. 2008] can be used to further scale concurrent shape analysis. Their approach is an extension of our approach, where parts of the global state are modeled separately.

Technically speaking, our framework is a generalization of [Sagiv et al. 2002] 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 explicit-state 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 [Nielson et al. 2000], which even allows the program to modify itself to support the semantics of Mobile Ambients [Cardelli and Gordon 1998].

1.1.2 *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 [Netzer and Miller 1992]). 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 [Michael and Scott 1996], also shown in Fig. 14 (b)) 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 ([Vermeulen 1997]) (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 [Lea 1997], 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 [Michael and Scott 1996] which are in part implemented in the `java.util.concurrent`

package of JDK1.5. (a preliminary version of this case study appeared in [Yahav and Sagiv 2003]).

For example, Fig. 2(a) shows a concurrent program using a queue. The implementation of the queue is given in Fig. 2(b) and Fig. 3. 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).

Our framework has been also applied to prove the correctness of the apprentice challenge, originally presented by J. Moore as a challenge for Java verification [Moore and Porter 2002].

Illegal Thread Interactions: The Java compiler does not prevent the programmer from introducing 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 in an `IllegalThreadStateException` being thrown. Our framework has been used to detect such illegal interactions.

1.1.3 *Prototype Implementation.* We have implemented a prototype of our framework called TVLA/3VMC [Yahav 2000]. In Section 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 Rinetskey and Sagiv [2001].

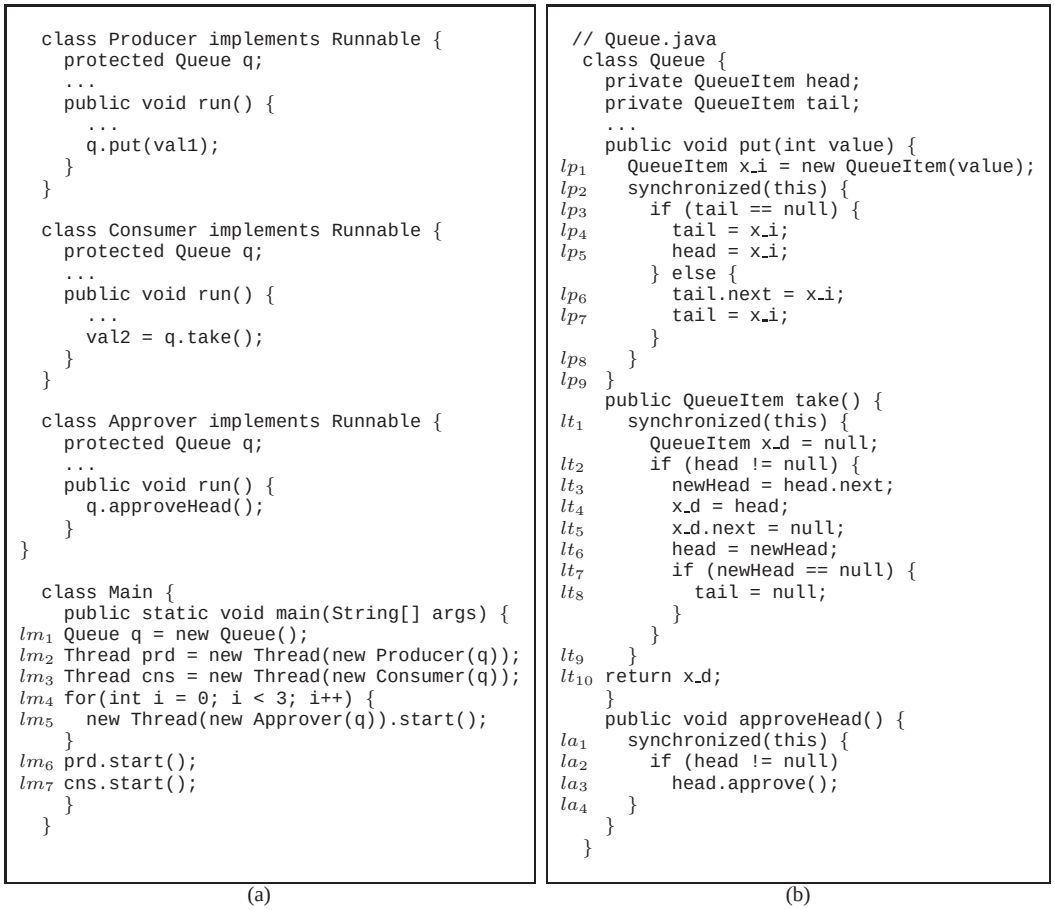
The examples used in this paper have been manually modeled as TVLA/3VMC files. It is possible to translate Java programs directly to TVLA by using a Soot-based [Vallée-Rai et al. 1999] front-end for Java developed by R. Manevich.

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.

While only being able to handle small programs, the framework is useful in practice when handling small but intricate concurrent heap-manipulating programs such as concurrent garbage collection algorithms [Vechev et al. 2007], and concurrent data structures [Vechev and Yahav 2008; Amit et al. 2007; Berdine et al. 2008].

In addition, our framework is flexible and powerful and can be used for prototyping analyses that can be later implemented in a more efficient manner.

1.1.4 *Paper Outline.* In Section 2, we give a brief overview of Java's concurrency model. Section 3 defines our formal model which uses logical structures to represent program configurations. Section 4 shows how multiple program configurations can be conservatively represented using a 3-valued logical structure. In Section 5, we show how our method can be used to detect several common concurrency errors. In Section 6, we describe the prototype implementation and the results we have obtained with it for a few small but interesting programs. In Section 7, we show how to apply our framework to verify correctness properties of implementations of concurrent queue algorithms. In Section 8, we apply our framework to verify the Apprentice Challenge. In Section 9, we survey closely related work. Finally, Section 10 concludes the paper and discusses future work.



```
  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)

Fig. 2. (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() {
     ...
   }
 }
```

Fig. 3. Simplified Java source code for a QueueItem implementation.

2. JAVA CONCURRENCY MODEL

We now give a brief description of the Java-like concurrency-primitives used in this paper. The reader is referred to [Gosling et al. 1997; Lea 1997; Lindholm and Yellin 1997; Goetz et al. 2006] 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.

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

The constructor for `Thread` class takes an object implementing the `Runnable` interface as a parameter. The `Runnable` interface requires that the object implements the `run()` method.

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 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 monitor associated with it, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor. In addition, each object has an associated block-set and wait-set for managing threads that are blocked on the object's monitor or waiting on it. When a `synchronized(expr)` statement is executed by a thread t, the expression $expr$ is evaluated, and the resulting object's monitor is checked for availability. If the monitor has not been *acquired* by any other thread, t *successfully acquires* it. If the monitor has already been acquired by another thread t', the thread t becomes *blocked* and is inserted into the monitor's block-set. A thread may acquire more than one monitor, and may acquire a monitor more than once (monitors are re-entrant). When a thread leaves the `synchronized` block, it *unlocks* the monitor associated with it. When a monitor has been locked more than once (by the same thread), it is released only when a matching number of *unlock* operations are performed.

In the example shown in Fig. 2, we guarantee that the queue operations are atomic by putting critical code into a `synchronized(this)` block.

A thread t can wait on an object o by calling the method `o.wait()`. Invoking `o.wait()` places t in o's wait-set, and releases the monitor lock associated with o. However, it does not release any other locks that t acquired. When a thread is in the wait-set of an object, we say that the thread is *waiting*. A *waiting* thread t can be only released by another thread invoking `o.notify()`, `o.notifyAll()` 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 [Buhr et al. 1995]. 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 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 from the actual details of the memory model and is common to most Java verification frameworks. While our framework is expressive enough for expressing the lower-level semantics involving the actual memory-model, this would result in a significant performance decrease.

—For simplicity, 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.

3. A PROGRAM MODEL

In this section, we lay the ground for our analysis framework. In Section 3.1, we use logical structures to represent the global state of a multithreaded program. Section 3.2 uses logical formulae as meta-language to extract interesting properties of a configuration, such as mutual exclusion. Then, in Section 3.3, we define a structural small-step operational semantics which manipulates configurations using logical formulae. Finally, in Section 3.4, we describe the safety properties that are verified in this paper.

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 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 blocked on a lock. Technically, first-order logic with transitive-closure is used in this paper 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 I contains the predicates used to analyze our example programs.

—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.

—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 successor relationship between integers.

—For every potential program location (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 local variable `fld`, there is a binary predicate $rv[fld](v_1, v_2)$ records the fact that the `fld` of the object v_1 points to the object v_2. For simplicity, we do not model the stack of a thread, and treat local variables of a thread as fields of the thread object.

—For every integer valued field `ifld`, there is a binary predicate $iv[ifld](v_1, v_2)$ 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 I are actually written in a generic way and can be applied to analyze different programs by modifying the set of labels and fields.

A (concrete) *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 heap object (some of which may represent threads of the program,). The configuration may also contain 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 [Sagiv et al. 2002]. The integrity constraints for integers are simply the Peano axioms encoded using FO formulae.

Predicates	Intended Meaning
$eq(v_1, v_2)$	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 I. Predicates for partial Java semantics.

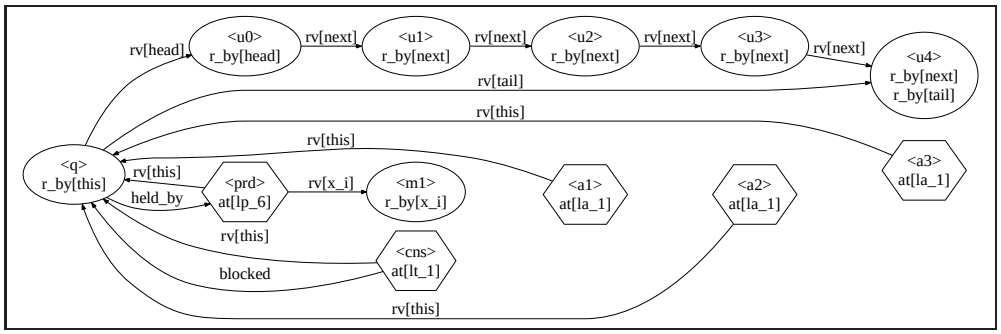


Fig. 4. A concrete configuration $C_4^\natural$.

In this paper, program configurations are depicted as directed graphs. Each individual of the universe is displayed as a node — objects of type thread are presented as hexagon nodes, other objects as round nodes. A unary predicate p which holds for an individual (node) u is drawn inside the node u. 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 a directed edge from u_1 to u_2 labeled with the predicate symbol. For brevity, predicate $eq(v_1, v_2)$ is not shown. We use a *natural* sign ($\natural$) to denote entities of the concrete domain (e.g., $C^\natural$ denotes a concrete configuration C).

EXAMPLE 3.1. The configuration $C_4^\natural$ shown in Fig. 4 corresponds to a global state of the example program with 5 threads: a single *producer* thread (labeled prd) that acquired the queue's lock, a single *consumer* thread (labeled cns) that is blocked on the queue's lock, and 3 *approving* threads ($a1, a2, a3$) that 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 heap allocated objects in a configuration is not bounded since the analyzed program may allocate new non-thread and/or thread individuals. We do not place a bound on the number of allocated objects.

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 and equality over the configuration. Appendix A provides a formal description of such formulae and their evaluation.

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

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

For ease of notation, we use the shorthand $\exists v\colon type.\varphi \triangleq \exists v.is\_type(v) \wedge \varphi$ (similarly for universal quantification). This allows us to write the above formula in a more readable form as:

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

For example, the formula

$$\exists t\colon thread.held\_by(l, t)$$

describes the fact that the lock l has been acquired by some thread. Our experience indicates that it is quite natural to express configuration properties using first-order logic.

Transitive closure is useful for expressing reachability. For example, to express the fact that an element u_1 in the queue q is reachable from `head` through a sequence of `next` fields, 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:

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

The above formula could be trivially extended to handle critical sections with multiple labels by using a disjunction of the labels in the critical section. It can also be extended to handle threads with different critical sections.

3.3 A Structural Operational Semantics of Configurations

Fig. 5 shows a depth-first search algorithm for exploring a state-space. 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

```
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'}
    }
  }
}
```

Fig. 5. State space exploration.

exploration used in model-checking [Clarke et al. 1999a]. 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 4.

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

—The *precondition* under which the action is enabled expressed as a 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.

—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 II 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 formula are assumed to remain unchanged by the action. The set of actions is partitioned into 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.

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

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

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

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

In essence, the predicates defined in this section, and the predicate update formulae we describe here, are an encoding of the concrete operational semantics. As such, it is up to the user to make sure that these formulae are a faithful representation of the operational semantics. We refer to the predicates that are used to encode the concrete operational semantics as *core predicates*. In Section 4, we will introduce the notion of instrumentation predicates that are used to refine the abstraction. The update formulae for these can be derived automatically using finite differencing [Reps et al. 2003]. The beauty of the approach of [Sagiv et al. 2002] (that we inherit here) is the fact that the predicate update formulae specified by the user are ones of the concrete semantics. The abstract transformers are automatically constructed by interpreting these formulae over 3-valued structures.

Formally, the meaning of actions is defined as follows:

DEFINITION 3.2. *Given a configuration $C^\natural$ and an action ac, we say that $C^\natural = \langle U, \iota\rangle$ **rewrites into a configuration** $C^{\natural\prime} = \langle U, \iota'\rangle$ (denoted by $C^\natural \Rightarrow_{ac} C^{\natural\prime}$), 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 II. We write $C^\natural \Rightarrow C^{\natural\prime}$ if for some action ac $C^\natural \Rightarrow_{ac} C^{\natural\prime}$.

In addition, there is a special action that creates a new individual u_{new} and results in a structure $C^{\natural\prime} = \langle U \cup \{u_{new}\}, \iota' \rangle$. A special predicate $isNew$ holds for u_{new}, and thus can be used in the predicate update formulae. The predicate $isNew$ is updated by the allocation action, and only holds (temporarily) for the newly allocated object(s). This predicate is required in order to distinguish newly allocated objects from objects that were pre-existing in a structure.

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 $0 \leq i < n$, $C_i^\natural \Rightarrow C_{i+1}^\natural$.*

3.4 Safety Properties of Java Programs

Given a set of initial configurations C_I, the set of *reachable* configurations C_R is the set of configurations that can be created by transitively rewriting a configuration from C_I. More formally, a configuration $C_r \in C_R$ iff there exists $C_i \in C_I . C_i \Rightarrow^* C_r$.

A safety property is formalized using a logical formula. 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 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 III 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. Note that 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 3.3. In Fig. 4, 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. 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 in this configuration 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 the 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

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
$\exists t : thread, l_1, l_2.blocked(t, l_1)$ $\wedge\, held\_by(l_2, t) \wedge \neg lt[id](l_2, l_1)$	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.
$\exists t.at[l_s](t) \wedge rv[var](t, l) \wedge \neg held\_by(l, t)$	Missing Ownership. Thread invoking `var.wait()` or `var.notify()` at label l_s when not holding the lock referenced by v.
See Section 5.2	Shared ADT
See Section 5.3	Thread Interactions

Table III. Violations of safety properties detected in this paper.

deadlock may be created when a thread is 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 `var.wait()` or `var.notify()` and does not hold the lock of the object l referenced by variable var.

4. AN ABSTRACT PROGRAM MODEL

The state-space exploration algorithm of Fig. 5 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 3 in a way that provides both feasibility and high precision.

In Section 4.1 we use 3-valued logical structures to conservatively represent multiple configurations of a multithreaded program. Section 4.1.1 presents the concept of embedding, which is crucial for proving the correctness of our algorithm. Section 4.2 presents the abstract semantics derived from the concrete semantics presented in Section 3.3. Finally, Section 4.3 shows how to improve the precision of our analysis by adding instrumentation predicates.

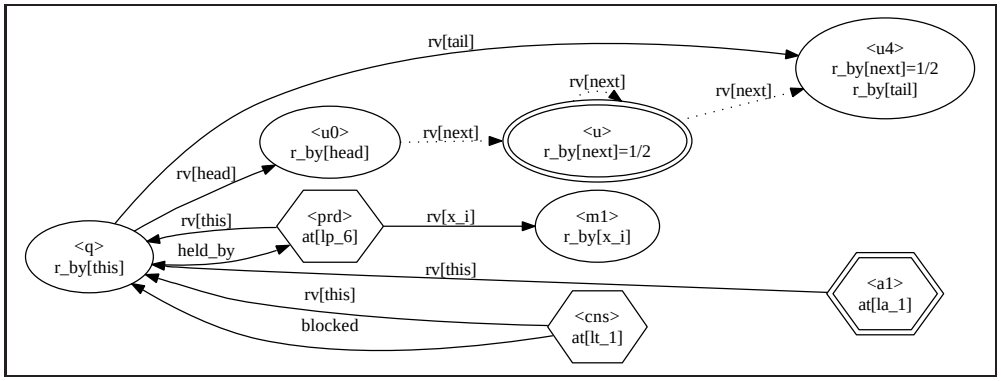


Fig. 6. An abstract configuration C_6 *representing* the (concrete) configuration $C_4^\natural$ of Fig. 4.

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 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 each predicate to its 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 some of the individuals represented by u have 1 as their truth values, and some have the truth value 0.

4.1.1 *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 into the same abstract individual.

Example 4.1.1. The abstract configuration C_6 *represents* concrete configuration $C_4^\natural$.

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 u represents all queue items that are not directly referenced by the queue's head or tail. Note that the abstract configuration C_6 represents many configurations. For example, it represents any configuration with 3 or more queue items (as u_0 and u_4 represent exactly one item each, and the summary node u represents at least one item). In a similar fashion, the abstract configuration represents configurations with one or more threads that reside at label la_1 (represented by the summary node labeled a_1). Note that the RW-Interference condition evaluates to 0 over the abstract configuration C_6.

The abstraction mechanism we describe here operates on a configuration as a whole. This may have obvious limitations on scalability as it uniformly applies the same abstraction to an entire configuration. Alternative approaches include separation and heterogenous abstraction [Yahav and Ramalingam 2004] applying different abstractions to different parts of a configuration, and heap decomposition [Manevich et al. 2008].

4.2 An Abstract Semantics

We use the same simple algorithm from Fig. 5 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* or best abstract transformer of an action) [Cousot and Cousot 1979] 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 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 III, without producing *false alarms* on our example programs.

Intuitively, our approach uses partial concretization (an operation called *focus* in [Sagiv et al. 2002]) to produce a finite set of abstract configurations to which the update is applied. The update is followed by an abstraction (*blur* in [Sagiv et al. 2002]) that produces the set of abstract configuration that constitute the result of the action.

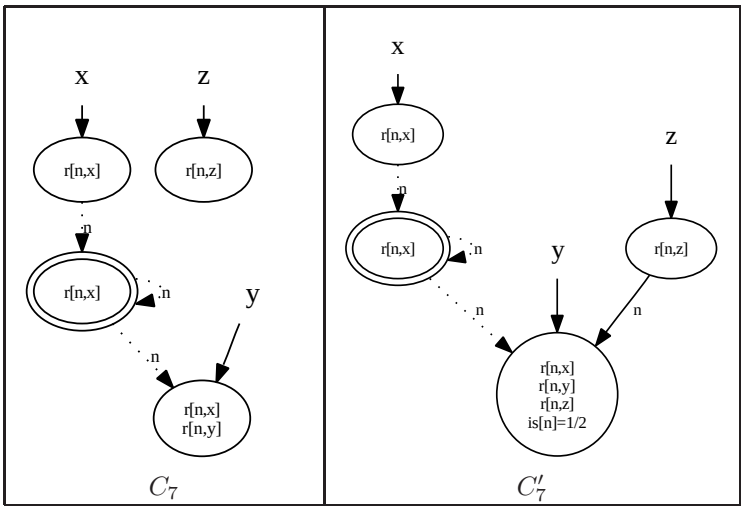


Fig. 7. Example of TVLA/3VMC transformer being more conservative than the best transformer.

In Section 6, we show that our abstract transformers are precise enough to verify the properties of interest in our example programs. However, our transformers may yield results that are more conservative than the best transformer, as shown in the following example.

Example 4.2.1. As a simple example of where our abstract transformers are more conservative than the best transformer, consider the abstract configurations shown in Fig. 7. For simplicity, we show abstract configurations of a sequential program in which there are no thread nodes. In these configurations, the program variables x,y, and z are represented using unary predicates and the field n using a binary predicate. In addition, we use the predicates $r[n,x]$,$r[n,y]$, and $r[n,z]$ to record transitive reachability from variables, and a predicates is to record whether a node is shared (pointed to by more than a single n field). Given the configuration C_7, we consider the effect of a single statement `z.n = y`. Applying the action corresponding to this statement to the abstract configuration C_7 results in the abstract configuration C'_7. Note that in the configuration C'_7, the value for the is predicate is $1/2$.

Applying the statement `z.n = y` to the abstract configuration C_7 makes the node pointed to by y become a shared node, as it is transitively reachable from the node pointed-to by x and (directly) reachable from the node pointed-to by z. Technically, this means that the value of the is predicate after the update should have been 1. However, our abstract transformer in this case is conservative and sets the value of is to $1/2$.

DEFINITION 4.1. *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 $C^{\natural'} = \langle U^\natural, \iota^{\natural'} \rangle$ such that:(i) $C^\natural$ is in the concretization of C, i.e., C* represents *$C^\natural$, (ii) C' is the* canonical abstraction *of $C^{\natural'}$, (iii) there exists an assignment Z that*

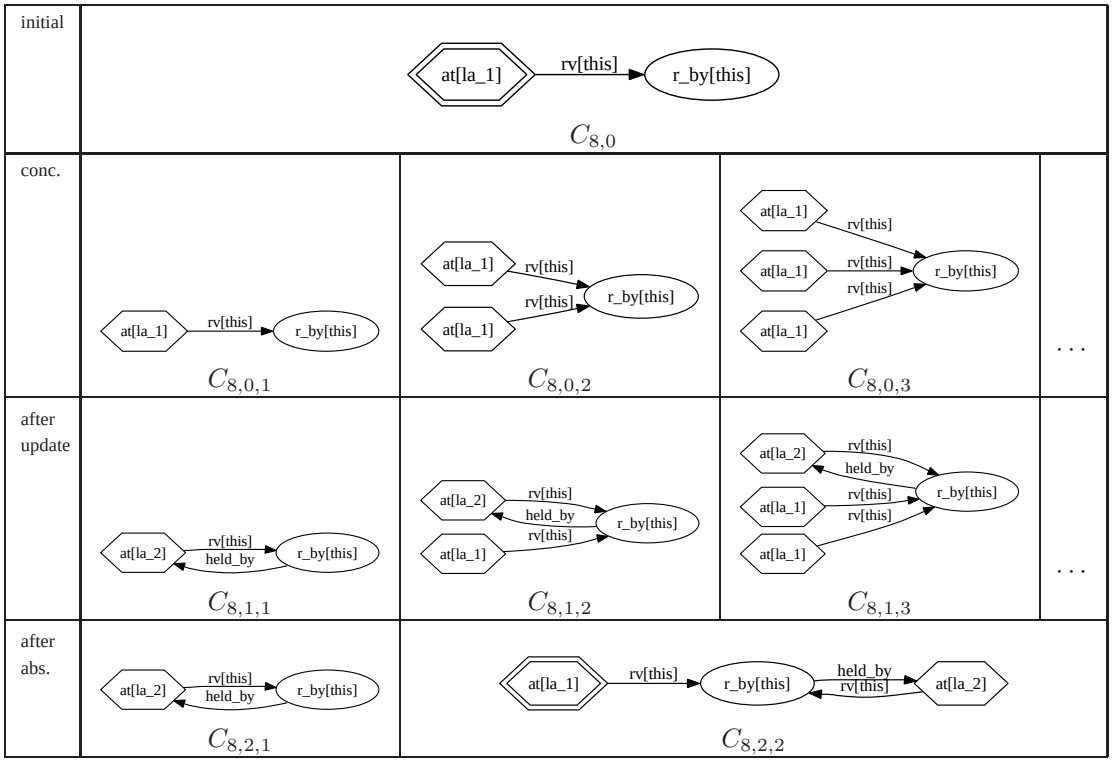


Fig. 8. Concretization and predicate-update for an unbounded number of threads all performing the $approveHead()$ method of the running example.

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 II, and* $[\![\varphi]\!]_3^C(Z)$ *is the three-valued evaluation of a formula* φ *in a configuration* C *under an assignment* Z *(see Appendix A). We write* $C \Rightarrow C'$ *if for some action* ac $C \Rightarrow_{ac} C'$*.*

Example 4.2.2. The abstract configuration $C_{8,0}$ shown in Fig. 8 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 $\{C_{8,0,1}, C_{8,0,2}, \ldots\}$ 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 $\{C_{8,1,1}, C_{8,1,2}, \ldots\}$ is the set after the application of $lock(v)$. The set of abstract configurations $\{C_{8,2,1}, C_{8,2,2}\}$ is the finite set of configurations after abstraction.

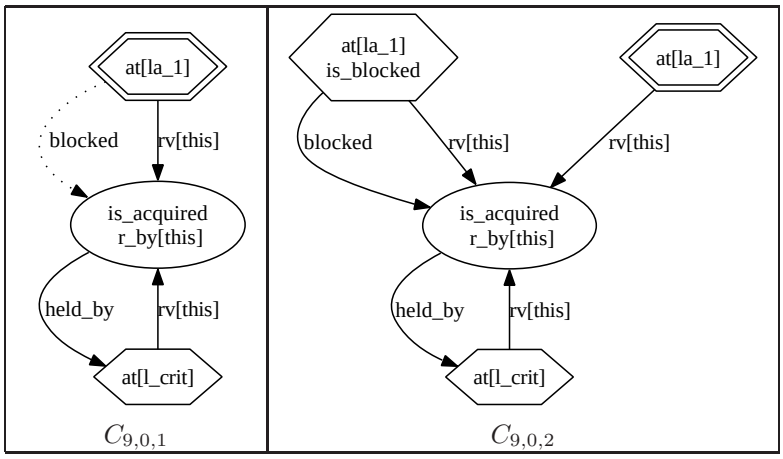


Fig. 9. Instrumentation predicate $is\_blocked(t)$.

The membership operator $member(C, stateSpace)$ of Fig. 5 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.

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 [Sagiv et al. 2002].

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.

Table IV shows some of the instrumentation predicates we used in this paper. We elaborate on the use of these predicates in Section 5. The following provides a simple example of their effect.

Example 4.3.1. 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 $C_{9,0,1}$ shown in Fig. 9 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 $C_{9,0,2}$ shows the same global state when the instrumentation predicate $is\_blocked(t)$ is used. 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.

4.4 Updating Instrumentation Predicates

The updated value of a core predicate is obtained by interpreting the corresponding update formula, specified in the concrete semantics, using 3-valued logic. The soundness of the abstract transformers updates to core predicates is guaranteed by construction.

The immediate question is how to update the value of instrumentation predicates?

As mentioned earlier, an instrumentation predicate is defined using a logical formula over core predicates. It is therefore possible to obtain the value of an instrumentation predicate after an update by re-evaluating its defining formula in the updated abstract configuration. However, as shown in [Sagiv et al. 2002], it is possible to achieve better precision by defining an update formula describing the effect of a transformer on an instrumentation predicate.

Reps et al. [2003] provide a method for updating instrumentation predicates automatically based on finite differencing. Their approach is able to derive the update formula for an instrumentation predicate under a transformer from the updates applied to the core predicates used to define it. This approach, however, is limited when it comes to updating instrumentation predicates using transitive closure (e.g., transitive reachability of Table IV). This is due to the inherent difficulty in incremental maintenance of transitive properties in directed graphs [Immerman 1998]. Still, the approach provides sufficiently precise updates in some special cases (e.g., acyclic graphs).

Technically, the current implementation also has limitations when dealing with allocation and deallocation, so update formulae for instrumentation predicates have to be provided in these cases. This is likely to be addressed in future versions of the tool.

5. VERIFYING SAFETY PROPERTIES

We use the instrumentation predicates listed in Table IV to improve the precision of our analyses. The following sections list more precise formulations of the formulae of Table III using instrumentation predicates whenever possible.

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 [Silberschatz and Galvin 1994].

Note that we are recording the order between lock `id`s and not the actual values of these `id`s. Thus, there is no requirement that the number of locks would be a priori bounded.

The formula for nested-monitors deadlock is given below:

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

Intuitively, a nested monitors deadlock occurs when a thread is waiting for an inner monitor to be released while holding the lock on an outer monitor that prevents access of other threads to the inner one.

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	$\exists o_t.rv[ref](t, o_t) \wedge rv[fld]^*(o_t, o)$

Table IV. Instrumentation predicates for partial Java semantics.

Technically, the formula above captures a situation in which there exists a thread t_w that is waiting on an inner monitor o_{in}, and is holding an outer monitor o_{out}, such that the inner monitor is reachable from the outer one, and there is no other pointer-path to the inner monitor other than the paths from the outer one. That is, the outer monitor dominates the paths into the inner one.

5.2 Shared Abstract Data Types

We define a set of reachability predicates similar to the ones defined in [Sagiv et al. 2002]. 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.

Examples of the encoding of such invariants are shown in Table VII of Section 7.2.2. Note that these formulae use actual program labels to capture the notion of the *end of an operation*.

5.3 Thread State Errors

We use additional predicates to record thread-state information: $ts\_created(t)$, $ts\_running(t)$, $ts\_blocked(t)$, $ts\_waiting(t)$ and $ts\_dead(t)$. These are core predicates that are updated directly by the instrumented semantics. In order to identify thread-state error properties, we add preconditions identifying when an action is illegal or suspicious. These preconditions are listed in Table V. Most of these properties (missing ownership

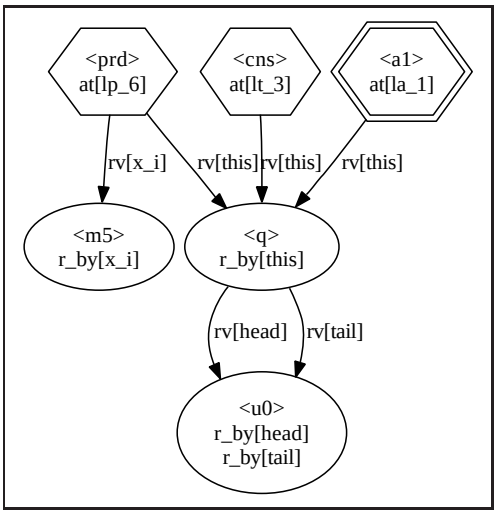


Fig. 10. An abstract configuration C_{10} in which interference between the consumer and the producer is detected.

properties are the exception) can be viewed as an explicit encoding of a typestate property ([Strom and Yemini 1986]) defining the permitted sequences of method calls for the type `java.lang.Thread`. The aforementioned thread-state predicates are used to encode the states of the typestate automaton. These predicates are a natural example of predicates used to record past events (e.g., $ts\_running(t)$ records the fact that the thread has been started). Similar predicates are used in [Shaham et al. 2003] to track typestate properties for compile-time memory management.

5.4 Interference

For simplicity, our formulation of interference assumes that we have statically classified program labels at which reads and writes occur. An alternative formulation would instrument the semantics to record reads and writes.

Example 5.4.1. Assume an erroneous version of the running example (Fig. 2) in which an unsynchronized version of `put()` is used. Configuration C_{10} shown in Fig. 10 demonstrates a possible interference in the program identified by our analysis. In the configuration C_{10} a consumer is trying to `take()` the last item, and a producer is simultaneously trying to `put()` an item.

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 III evaluates to 1 for this configuration since both threads reference the same object $\langle u0 \rangle$. 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 correctly report that RW-Interference does not occur (since `head` and `tail` refer to different objects).

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

Table V. Preconditions for checking illegal and suspicious thread interactions.

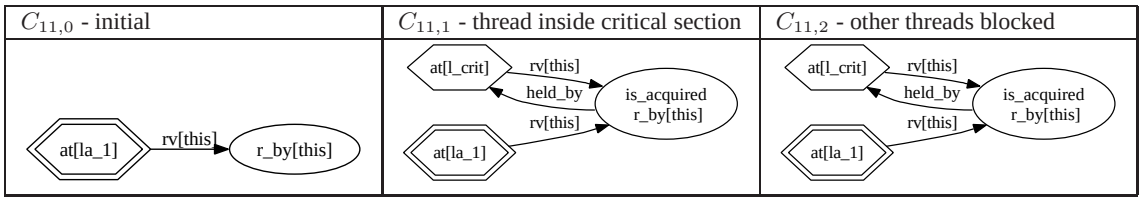


Fig. 11. Configurations arising in mutual exclusion with an unbounded number of threads.

5.5 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. In [Emerson and Sistla 1993], 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 [Emerson and Sistla 1993], and verifying that the *mutual exclusion* property holds for an *unbounded number of threads*.

Example 5.5.1. 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 example 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. 11 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

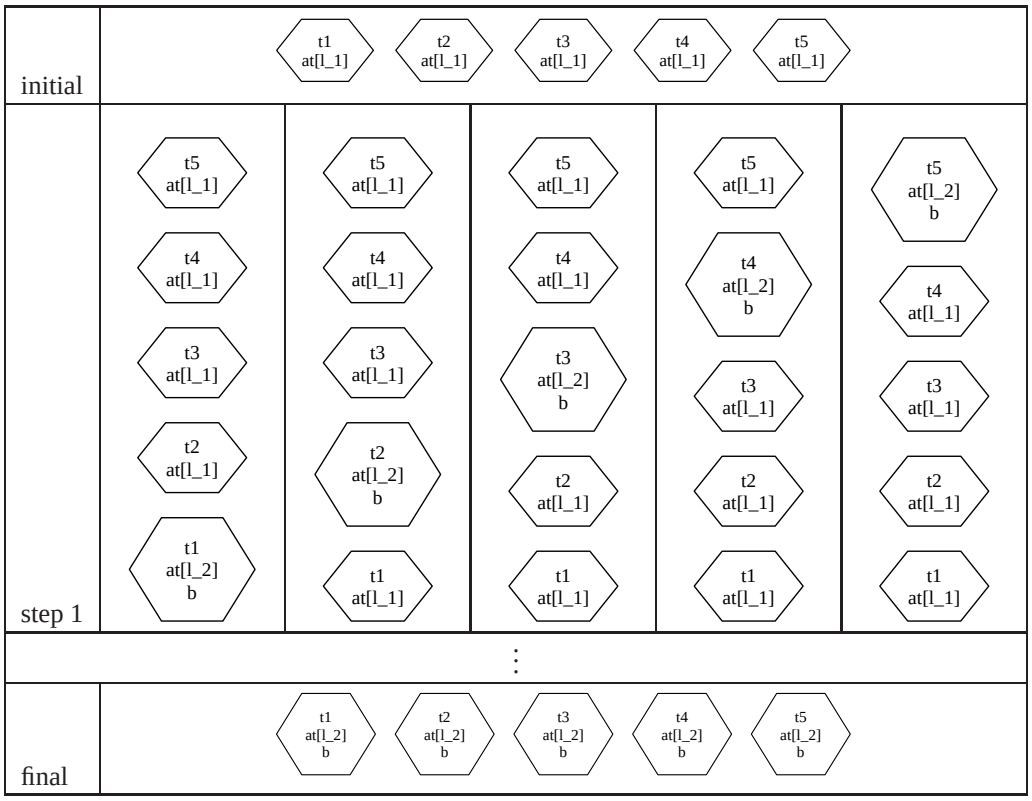


Fig. 12. Configurations arising with explicit thread names.

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 in 125 possible interleavings that have to be considered (see Fig. 12). Analyzing the program in our approach will only consider a single (representative) interleaving (see Fig. 13).

6. PROTOTYPE IMPLEMENTATION

In this section, we briefly describe our prototype implementation and present experimental results for 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 the following sections.

We have implemented a prototype of our framework called TVLA/3VMC [Yahav 2000]. Our implementation is based on the 3-valued logic engine of TVLA [Lev-Ami and Sagiv 2000]. We applied the analyses to several small but interesting programs. Table VI summarizes the programs we tested with: number of configurations created, actual errors (AE), and reported errors (RE). All analyses terminated in less than 2 hours.

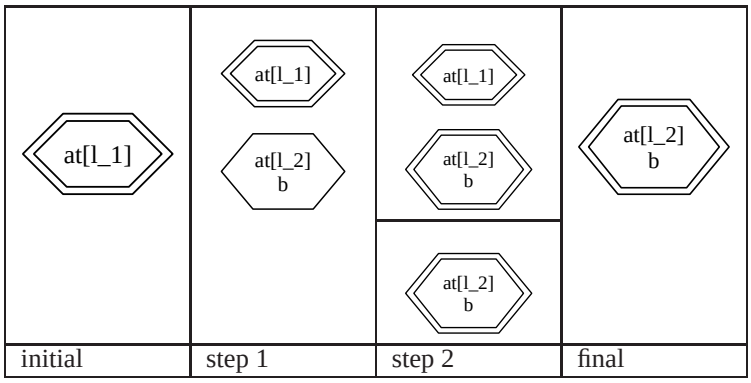


Fig. 13. Configurations arising with canonical thread names.

It is important to note that the cost of verification for an unbounded number of threads in our approach is exponential in the number of predicates, while the cost of verification with explicit 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 [Sagiv et al. 2002] for more details).

The $swap$ and $swap\_ord$ programs use two threads to swap 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 that uses mutual exclusion to protect a critical section. $prodcons$ and $sProdCons$ are implementations of a Queue ADT manipulated by producer and consumer threads. DP is an implementation of the *dining philosophers* problem with unbound 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 terminated with no false alarms. In the following sections, we explore more realistic example programs.

7. 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.

7.1 Concurrent Queue Algorithms

Concurrent FIFO queues are widely used in concurrent systems. Queues are used in scheduling mechanisms, and as a basis of many concurrent algorithms. Concurrent manipulation of a shared queue requires synchronization to guarantee consistent results.

Program	Description	Properties	Config.	AE/RE
swap	swap elements	data races and deadlock	25	1/1
swap_ord	swap elements with resource ordering	data races and deadlock	48	0/0
stack	non-synchronized stack	data races	184	1/1
sStack	synchronized stack	data races	104	0/0
mutex	mutual exclusion	mutex	41	0/0
nestedMon	nested monitors	deadlock	42	0/0
prodCons	producer consumer	data races	416	1/1
sProdCons	synchronized producer consumer	data races	195	0/0
DP	unbounded dining philosophers	deadlock	514	0/0

Table VI. Number of configurations, actual errors (AE), and reported errors (RE) for the programs analyzed.

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 [Michael and Scott 1996; Stone 1990; 1992; Prakash et al. 1991; Wing and Gong 1990; Vechev and Yahav 2008]. The algorithms in [Michael and Scott 1996; Stone 1990; 1992; Prakash et al. 1991] are given without a formal proof of correctness, and [Wing and Gong 1990] provides a manual formal proof.

We focus on the non-blocking queue and two-lock queue algorithms presented in [Michael and Scott 1996]. A Java-like code for the queue implementations is given in Fig. 14.

To emulate the intention of [Michael and Scott 1996], our programming model diverges from Java by assuming a free operation and supporting several operations defined below. The challenge of memory-management in such concurrent algorithms deserves a separate discussion that goes beyond the scope of this paper. The interested reader can find more details in [Michael 2004] and [Vechev et al. 2009].

In this section, we present the concurrent queue algorithms and the correctness properties we will verify for these algorithms.

7.1.1 *Non-Blocking Queue.* Java-like pseudo-code for the non-blocking queue algorithm is shown in Fig. 14(a). The queue uses an underlying singly-linked list which is pointed 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 was not updated by other threads. CAS was introduced on the IBM System 370 [ibm 1983]. It is supported on Intel and Sun SPARC processor architectures.

The CAS primitive takes 3 arguments — an address, an expected value, and a new value; it then atomically compares the value at the address to the expected value, and if the values

```
// Non Blocking Queue
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   QueueItem node = new QueueItem(value);
e2   node.value = value;
e3   node.next.ref = NULL;
e4   while(true) { //Keep trying until done
e5     QueueItem tail = this.Tail;
e6     QueueItem 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     QueueItem head = this.Head;
d3     QueueItem tail = this.Tail;
d4     QueueItem 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;
    ...
}
```

(c)

Fig. 14. Java-like pseudo-code for (a) non-blocking queue, (b) two-lock queue, (c) queue item.

are equal updates the address to contain the new value. If the value at the address is not equal to the expected value, no update is applied.

CAS-based algorithms may suffer from the "ABA" problem [Michael and Scott 1996] in which a sequence of read-modify-CAS results with a swap when it shouldn't. This happens when a thread t_1 reads a value A of a shared variable, computes a new value, and performs 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 that performs a comparison of the `ref` component of two reference variables.

It is worth noting that a variation of the algorithm that uses the synchronization primitives load-linked/store-conditional (LL/SC) will not require the modification counters. The CAS primitive is universal [Herlihy 1991] and used in common architectures and we therefore chose to focus on implementations using that primitive.

These algorithms can be simplified further by assuming a garbage collector instead of explicit memory management (see [Vechev et al. 2009; Vechev and Yahav 2008]).

7.1.2 *Two-Lock Queue.* Fig. 14(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.

7.1.3 *Correctness of Algorithms.* The correctness of the queue algorithms in [Michael and Scott 1996] 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.

We note that these properties are not the only properties required for showing that the queue algorithms are indeed correct. Ideally, we would like to automatically verify that the queue algorithms are linearizable [Herlihy and Wing 1990]. Indeed, recently, [Amit et al. 2007; Berdine et al. 2008; Vafeiadis 2009] automatically proved the linearizability of these algorithms, but this requires techniques that are beyond the scope of this paper. In addition, we do not address liveness properties of these algorithms. Gotsman et al. [2009] provide a nice discussion of liveness properties for these algorithms and a technique for their automatic verification.

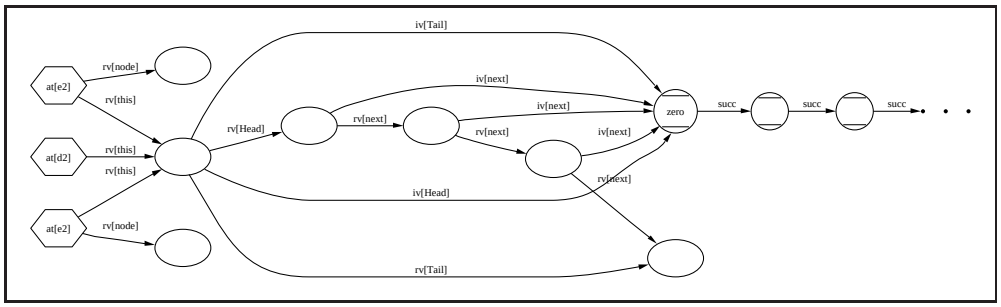


Fig. 15. A concrete configuration $C_{15}^{\natural}$ with two enqueueing and one dequeueing threads.

In this paper, we focus on proving the above structural properties that are still rather challenging to verify automatically. In the following sections, we formally state these claims, and automatically verify them using TVLA/3VMC.

7.2 Vanilla Verification Attempt

In this section, we describe the basic steps required to verify the concurrent queue algorithms using TVLA/3VMC.

7.2.1 *Representing Program Configurations using First-Order Logical Structures.* We now show how to apply our technique to verify the concurrent queue algorithms.

The non-blocking queue algorithm uses unsigned integer values as reference time-stamps. As described in Section 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 7.3 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 $C_{15}^{\natural}$ shown in Fig. 15 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.

7.2.2 *Safety.* The first step in verifying the properties of Section 7.1.3 in TVLA/3VMC is to formulate them in FO^{TC} using the predicates defined in Table I.

The immediate question is *when* can the properties be checked. One alternative is to phrase the properties such that they are global invariants that hold for all configurations of the queues. Another is to check that the invariants hold when the queue is "stable", that is, no operation is currently executing. The latter is similar to checking quiescence-consistency [Herlihy and Shavit 2008], and is the one we choose here. Table VII provides the formulation of the properties P1-P5 for the non-blocking queue algorithm. The

	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 VII. Safety properties for the non-blocking queue algorithm.

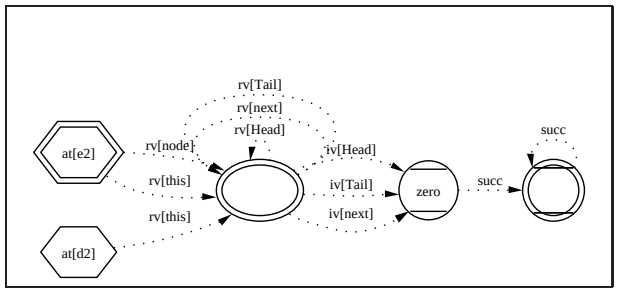


Fig. 16. An abstract configuration C_{15} representing the concrete configuration $C_{15}^{\natural}$ of Fig. 15.

formulation of these properties for the two-lock queue only differs in label names. For each property defined informally in Section 7.1.3, we provide a corresponding formula in FO^{TC}.

Properties P2-P3 are being checked at the end of their corresponding operations, and assume that the queue is stable (i.e., no other operation is executing concurrently). Properties P1, P4, and P5 assume that no queue operation is in progress. We use flags to determine when operations are ongoing (not shown here for simplicity).

In the table, we use the shorthand nbq to abbreviate `NonBlockingQueue`. Formula P1 uses transitive reachability from `Head` to require the queue tail is reachable from the queue head—thus the queue is always connected (existence of a tail element is guaranteed by requirement P5). 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 to check it. 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 requires that there is no queue element u such that it precedes the head of the queue. Finally, formula P5 ensures that a tail element exists.

7.2.3 *Abstraction*

Example 7.2.2. The abstract configuration C_{15} shown in Fig. 16 is obtained by applying canonical abstraction to the concrete configuration $C_{15}^{\natural}$ of Fig. 15.

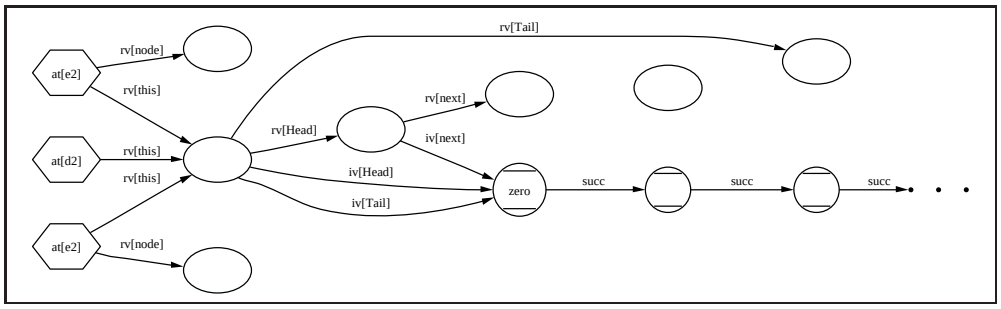


Fig. 17. A concrete configuration $C^{\natural}_{15,1}$ that is embedded in C_{15} and violates queue connectedness (property P1).

The summary thread node represents the two enqueueing threads of the concrete configuration $C^{\natural}_{15}$, 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 threads that are at their initial labels.

Unfortunately, this abstract configuration also represents the concrete configuration $C^{\natural}_{15,1}$ which violates the connectedness property (P1), meaning that we fail to verify that P1 holds. Indeed, since each subformulae of P1's body evaluates to $1/2$ over the abstract configuration C_{15}, using Kleene evaluation of boolean operators yields the value $1/2$ for P1. In the next section, we will describe a way to remedy that.

7.3 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 [Graf and Saidi 1997]. 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 the same distinctions can be used for many programs, and furthermore, these distinctions correspond to fundamental properties of data-structures (e.g., sharing, reachability). This paper confirms this by showing that the standard set of distinctions suffices for verifying all the desired properties of the concurrent queue algorithms.

Technically, refining the abstraction is achieved by introducing the unary predicates of Table VIII. 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., [Shaham et al. 2003]), however, for this case study we just use the standard TVLA/3VMC instrumentation predicates.

Predicates $rt[fld, next](t, o)$ allow us to track reachability information of items inside the queue. For example, the instrumentation predicate $rt[Head, next](v)$ may be used

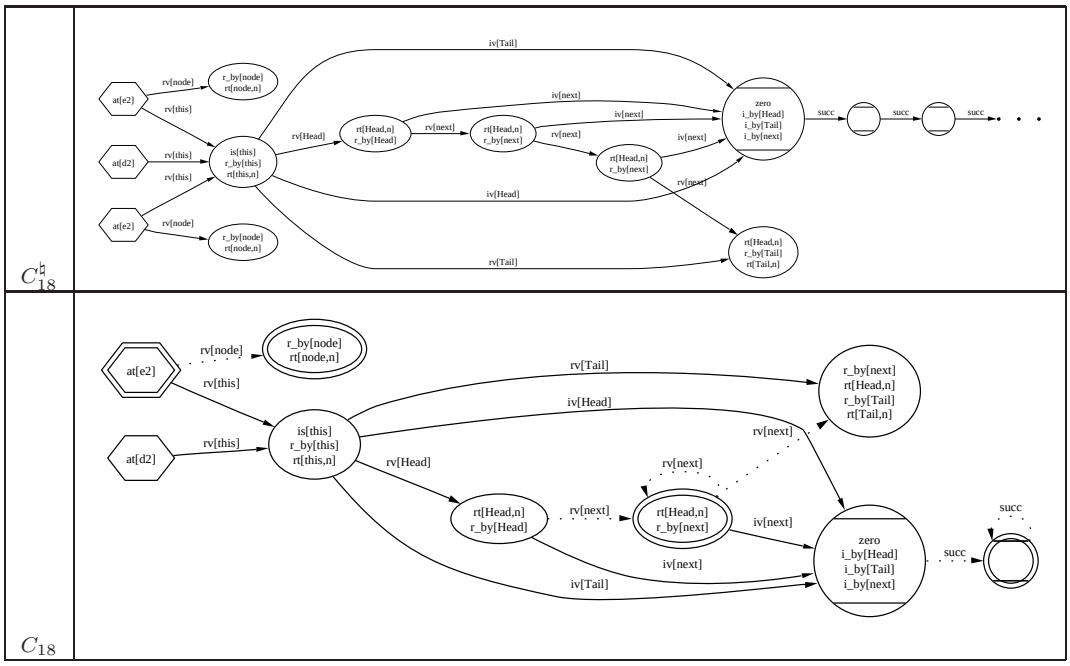


Fig. 18. Concrete configuration $C_{18}^{\natural}$ using instrumentation predicates, and its canonical abstraction C_{18}.

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	$\exists o.iv[fld](o,n)$
$is[fld](o)$	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	$\exists t.held\_by(l,t)$
$rt[fld,next](o)$	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 VIII. Instrumentation predicates used in our example program.

to track reachability of items from the head of the queue using a path of *next* references. These predicates are an adaptation for multi-threaded programs of the reachability instrumentation predicates presented in [Sagiv et al. 2002]. Similarly, predicates $is[fld](o)$ are an adaptation of sharing predicates of [Sagiv et al. 2002]. The predicates $is\_acquired(l)$ and $r\_by[fld](l)$ were discussed in Section 4.3. 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.

Once a collection of instrumentation predicates is defined, we have to specify how these predicates are updated by program actions. Update formulae for the instrumentation pred-

Program	Description	Configs
nbq_enqueue	unbounded number of enqueue-ing threads	1833
nbq_dequeue	unbounded number of dequeue-ing threads	1098
nonblockq_err1	err - negated condition at e8	36
nonblockq_uni	err - start with uninitialized queue	17
tlq_enqueue	unbounded number of enqueueing thrads	982
tlq_dequeue	unbounded number of dequeuing threads	225
twolockqn	single producer and single consumer	975
twolockq_err1	err - broken producer synchronization	24

Table IX. Number of configurations explored by analysis of the queue algorithms.

icates used in this case study were supplied manually due to technical limitations of automatic derivation using finite differencing [Reps et al. 2003].

Subformulae of the safety properties are replaced with the corresponding instrumentation predicate to improve precision.

Example 7.3.1. Fig. 18 shows the concrete configuration $C^{\natural}_{18}$ which is an instrumented version of $C^{\natural}_{15}$, and its canonical abstraction C_{18}. The additional information recorded by the instrumentation predicates $rt[Head, next](v)$ and $rt[Tail, next](v)$ allows us to observe that queue connectedness (property P1) is maintained in the abstract configuration C_{18} since P1 evaluates to 1. Moreover, this implies that concrete configurations of the form of $C^{\natural}_{15,1}$ are no longer represented.

7.4 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.

Table IX presents the analysis results for variations of the concurrent queue algorithms. All analyses terminated in less than 2 hours.

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. Our prototype reported errors in both cases.

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.

In addition, TVLA/3VMC does not yet benefit from the latest (experimental) improvements implemented in TVLA [Bogudlov et al. 2007].

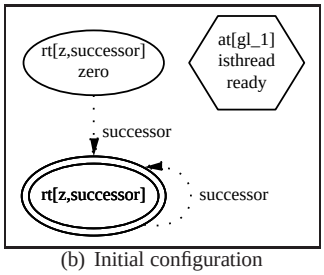
A fundamental question in program analysis is how to predict the precision of a given analysis on a given program. In principle, this is a hard question. In our setting, we note that the abstraction of TVLA/3VMC incurs a significant loss of precision when the safety

```
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) {
gl_1: Container container = new Container();
      for (;;) {
         Job job = new Job();
         job.setref(container);
         job.start();
      }
   }
}
```

(a) Apprentice Challenge Source.



(b) Initial configuration

Fig. 19. Source code and initial abstract configuration for the Apprentice Challenge.

of the verified program depends on arbitrary arithmetic operations on integer variables. It is possible to address this loss of precision by integrating our heap abstraction with a more powerful numerical abstraction (e.g., [Gopan et al. 2005]).

8. 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.

8.1 Problem Statement

The apprentice challenge was presented by Moore [Moore and Porter 2002] 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. 19(a) increases monotonically (under all possible schedules).

8.2 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 a simplified bounded version of the problem (i.e., the "finite Apprentice").

In our solution, we use the predicates described earlier in Section 3 and Section 7.3. 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 integers). For simplicity, we do not introduce such terminating node and assume that integers may increase infinitely.

The initial configuration for the apprentice challenge is shown in Fig. 19(b). In this configuration, there is a single thread node, corresponding to the main program thread. This thread is at the initial label $\texttt{gl}_1$, and is ready to be scheduled. The other nodes in the configuration represent integer values: one node represents the value zero, and the summary node summarizes the rest of the integer values.

In order to show that the counter increases monotonically, our model records the value of the `counter` variable on entry to `incr()`. Technically, this can be thought of as using a two-vocabulary structure (see e.g., [Jeannet et al. 2004]).

8.3 Results

We applied TVLA/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.

The techniques used in [Moore and Porter 2002] are different than what we use here, and they also use a different machine setup for experiments. Therefore a direct comparison of the running times is not appropriate. However, at least for this example program, we believe that our approach requires less human effort and fewer computation resources.

We have also applied TVLA/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 and taking 13.8 MB of memory.

Unlike the ACL2 solution for the apprentice challenge [Moore and Porter 2002], our approach is based on a conservative abstraction of the concrete Java semantics. Generally, this means that we might produce false alarms even when a property does hold for the verified program. However, for the Apprentice challenge, we are able to verify the goal property with no false alarms.

9. RELATED WORK

In this section, we provide a brief survey of closely related work from the areas of shape analysis and model checking.

Shape Analysis of Concurrent Programs

Shape analysis has been an active research topic for over 30 years. Here, we focus our discussion on shape analysis specifically aimed at concurrent programs, and do not discuss the large volume of work on shape analysis for sequential programs (e.g., [Jones and Muchnick 1981; 1982; Chase et al. 1990; Sagiv et al. 1998; Moller and Schwartzbach 2001; Balaban et al. 2005; Berdine et al. 2005; Distefano et al. 2006; Berdine et al. 2007; Zee et al. 2008; Yang et al. 2008; Calcagno et al. 2009]). The reader is referred to [Sagiv et al. 2002; Reps et al. 2004] and [Rinetzky 2008] for discussion of work related to shape analysis of sequential programs.

Thread-Modular analyses. The thread-modular approach of Flanagan et al. [2002] performs assume-guarantee (modular) verification for multi-threaded programs. In principle,

this modular approach can scale well, as it verifies one thread at a time. However, the approach relies on user-specified environment assumptions that may be challenging to obtain. Combining our approach with a heterogenous abstraction like the ones in [Yahav and Ramalingam 2004] may provide an automated means for computing environment invariants.

Leino and Müller [2009] present an approach for modular verification of concurrent object-based programs that is based on dynamic frames and fractional permissions. The basic idea is similar to [Bornat et al. 2005; O'Hearn 2007] (see below), but is specialized to object-based programs, and uses verification conditions in first-order logic rather than separation logic.

Analyses based on 3*-valued logic.* Our work provides the basis for concurrent shape analysis using the 3-valued logic framework of Sagiv et al. [2002].

In [Amit et al. 2007], the abstractions and tools presented in this paper are extended to verify linearizability [Herlihy and Wing 1990], a main correctness condition of concurrent data structures, for a fixed number of threads.

In [Berdine et al. 2008; Manevich et al. 2008], thread quantification and heap decomposition are used to analyze programs with an unbounded number of threads. Thread quantification adds an extra level of universal quantification to enable analyzing programs with an unbounded number of threads and heap decomposition is used to abstract away unnecessary correlations between resource invariants and local thread states to obtain scalability. These techniques help the analysis scale and enable the verification of linearizability with an unbounded number of threads in challenging programs.

Separation-logic based analyses. Separation logic [Ishtiaq and O'Hearn 2001; Reynolds 2002] has been used as a basis for concurrent shape analysis.

Concurrent separation logic [O'Hearn 2007] allows to manually verify race-free heap-manipulating programs by associating a resource invariant with every thread-shared subheap. It uses the insight that parts of shared memory are often protected by locks that guarantee mutual exclusion. When a thread obtains the protecting lock of a subheap, it owns the subheap, and other threads cannot access it (some refinements of this idea to allow concurrent reads have been investigated e.g., using fractional permissions [Bornat et al. 2005]).

Gotsman et al. [2007], employ the idea of associating shared-resources with invariants to present a thread-modular shape analysis that leverages locks to partition the heap into a (bounded) number of subheaps. Their approach requires a user-specified association between subheaps and the locks that protect them to partition the heap, and computes the resource invariants using a reachability-based heuristic.

Vafeiadis [2009] presents a "value abstraction" that extends the symbolic shape analysis of Distefano et al. [2006] by recording correlations between equal values. This abstraction is used in a static analyzer based on RGSep [Vafeiadis 2008] to automatically verify linearizability of challenging fine-grained concurrent algorithms.

Allocation-site based analyses. Corbett [2000] uses a simple shape analysis of concurrent Java programs to reduce their finite-state models. In this analysis, the number of threads is bounded. The algorithm presented is based on [Chase et al. 1990], 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).

There is a wide variety of approaches for static race detection (e.g., [Sterling 1993; Flanagan and Abadi 1999; Flanagan and Freund 2000; Boyapati and Rinard 2001; Choi et al. 2001; Boyapati et al. 2002; Engler and Ashcraft 2003; Flanagan and Freund 2004; Henzinger et al. 2004; Naik et al. 2006; Pratikakis et al. 2006; Flanagan et al. 2008]). Most of these approaches are based on allocation-site based abstraction of the heap. Broadly speaking, our approach does not scale as well as these approaches, but is able to verify more subtle cases of non-interference by using a finer abstraction of the heap.

Under Approximations. Lal and Reps [2008] present an approach for reducing concurrent analysis under a context bound to sequential analysis. Lahiri et al. [2009] use context-bounded analysis of concurrent programs based on an SMT solver to automatically detect errors in C programs. Their transformation from a concurrent program with a fixed context-bound into a sequential program is based on the translation of Lal and Reps [2008]. Their approach is fully automatic, and can handle a subset of C. Additional details about this line of work can be found in [Lahiri and Qadeer 2008; Atig et al. 2009].

Model Checking

Many approaches were proposed to handle model checking 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., [Strom 1983; Strom and Yemini 1986]). As a second phase, these works use one of the numerous approaches for model-checking concurrent finite-state programs (e.g., [Cook et al. 2005]), performing various forms of bounded model-checking (e.g., [Qadeer and Rehof 2005; Ganai and Gupta 2008]), and using predicate abstraction (e.g., [Das et al. 1999]).

In our framework, rather than having separate model-extraction and model checking phases, we follow the abstract-interpretation approach [Cousot and Cousot 1977] and cast our analysis in a syntax-directed manner. Other approaches use a combination of theorem-proving and model checking techniques to automatically construct such abstractions [Abdulla et al. 1999; Bensalem et al. 2000; Bensalem et al. 1998].

In the following, we briefly discuss some closely related work that addresses Java programs or employs some sort of abstraction.

Predicate-abstraction based model checking. Clarke et al. [2006; 2008] present a framework that extends predicate abstraction with ideas from *counter abstraction* [Pnueli et al. 2002], counting the number of threads that are in every (local) state of a process, similar to the way in which we abstract threads. We believe that our abstraction is more natural for expressing heap properties and interactions between threads and the heap (see discussion in [Manevich et al. 2005]).

Das, Dill, and Park [1999] use 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 [2000] 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.

Bounded model checking. JavaPathFinder [Havelund and Pressburger 2000] and Java2Spin [Demartini et al. 1999a] translate Java source code to PROMELA representation. The SPIN model-checker [Holzmann 1995] 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 [Demartini et al. 1999b] supports dynamic allocation of objects. However, since it uses no abstraction, it can only handle bounded data-structures and a bounded number of threads. Vechev et al. [2009] use SPIN for model checking linearizability of concurrent data structures. Clarke et al. [1997] present 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.

JavaFan [Farzan et al. 2004] is a framework for analyzing multithreaded Java programs based on the Maude rewriting system [Clavel et al. 2002]. It supports symbolic simulation of concurrent programs and bounded model checking. However, it does not use abstraction and cannot be used for verifying programs with an unbounded state space.

Stoller [2000] presents a framework for model checking distributed Java programs. 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.

10. CONCLUSION AND FUTURE WORK

We have presented a parametric framework for verifying safety properties of concurrent heap-manipulating programs. Our framework is a generalization of 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.

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 these techniques, we were able to automatically verify non-trivial properties of heap-manipulating programs that have not been automatically verified in the past.

In the future, we intend to exploit *partial order reduction* techniques such as [Valmari 1991; Godefroid 1996; Flanagan and Godefroid 2005; Gueta et al. 2007] in order to improve scalability of our analysis.

Acknowledgments

We thank the anonymous reviewers for their insightful comments which have significantly improved this manuscript. We thank Noam Rinetzky for comments on an earlier version of this manuscript.

REFERENCES

ABDULLA, P. A., ANNICHINI, A., BENSALEM, S., BOUAJJANI, A., HABERMEHL, P., AND LAKHNECH, Y. 1999. Verification of infinite-state systems by combining abstraction and reachability analysis. In *CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification*. Springer-Verlag, London, UK, 146–159.

AMIT, D., RINETZKY, N., REPS, T., SAGIV, M., AND YAHAV, E. 2007. Comparison under abstraction for verifying linearizability. In *Computer Aided Verification, 19th International Conference, CAV 2007*. Lecture Notes in Computer Science. Springer.

ATIG, M. F., BOUAJJANI, A., AND QADEER, S. 2009. Context-bounded analysis for concurrent programs with dynamic creation of threads. In *Tools and Algorithms for the Construction and Analysis of Systems*. Lecture Notes in Computer Science, vol. 5505. Springer, 107–123.

BALABAN, I., PNUELI, A., AND ZUCK, L. D. 2005. Shape analysis by predicate abstraction. In *Verification, Model Checking, and Abstract Interpretation*. LNCS, vol. 3385. Springer, 164–180.

BALL, T., MAJUMDAR, R., MILLSTEIN, T., AND RAJAMANI, S. 2001. Automatic predicate abstraction of C programs. In *Proc. Conf. on Prog. Lang. Design and Impl.* 203–213.

BENSALEM, S., GANESH, V., LAKHNECH, Y., MUÑOZ, C., OWRE, S., RUESS, H., RUSHBY, J., RUSU, V., SAÏDI, H., SHANKAR, N., SINGERMAN, E., AND TIWARI, A. 2000. An overview of SAL. In *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, C. M. Holloway, Ed. Hampton, VA, 187–196.

BENSALEM, S., LAKHNECH, Y., AND OWRE, S. 1998. Invest: A tool for the verification of invariants. In *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*. Springer-Verlag, London, UK, 505–510.

BERDINE, J., CALCAGNO, C., COOK, B., DISTEFANO, D., OHEARN, P. W., WIES, T., AND YANG, H. 2007. Shape analysis for composite data structures. In *Computer Aided Verification*. Lecture Notes in Computer Science, vol. 4590. Springer, 178–192.

BERDINE, J., CALCAGNO, C., AND OHEARN, P. W. 2005. Symbolic execution with separation logic. In *Programming Languages and Systems*. Lecture Notes in Computer Science, vol. 3780. Springer, 52–68.

BERDINE, J., LEV-AMI, T., MANEVICH, R., RAMALINGAM, G., AND SAGIV, M. 2008. Thread quantification for concurrent shape analysis. In *Computer Aided Verification, 20th International Conference, CAV 2008*. Lecture Notes in Computer Science, vol. 5123. Springer, 399–413.

BOGUDLOV, I., LEV-AMI, T., REPS, T. W., AND SAGIV, M. 2007. Revamping tvla: Making parametric shape analysis competitive. In *Computer Aided Verification, 19th International Conference, CAV 2007*. Lecture Notes in Computer Science, vol. 4590. Springer, 221–225.

BORNAT, R., CALCAGNO, C., O'HEARN, P., AND PARKINSON, M. 2005. Permission accounting in separation logic. *SIGPLAN Not. 40*, 1, 259–270.

BOYAPATI, C., LEE, R., AND RINARD, M. 2002. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM Press, New York, NY, USA, 211–230.

BOYAPATI, C. AND RINARD, M. 2001. A parameterized type system for race-free java programs. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*. ACM Press, New York, NY, USA, 56–69.

BUHR, P. A., FORTIER, M., AND COFFIN, M. H. 1995. Monitor classification. *ACM Computing Surveys 27*, 1, 63–107.

CALCAGNO, C., DISTEFANO, D., O'HEARN, P., AND YANG, H. 2009. Compositional shape analysis by means of bi-abduction. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, New York, NY, USA, 289–300.

CARDELLI, L. AND GORDON, A. D. 1998. Mobile ambients. In *FoSSaCS*, M. Nivat, Ed. Lecture Notes in Computer Science, vol. 1378. Springer, 140–155.

CHASE, D., WEGMAN, M., AND ZADECK, F. 1990. Analysis of pointers and structures. In *Proc. Conf. on Prog. Lang. Design and Impl.* ACM Press, New York, NY, 296–310.

CHOI, J., LOGINOV, A., AND SARKAR, V. 2001. Static datarace analysis for multithreaded object-oriented programs. IBM Research Report 22146, IBM Research.

CLARKE, E., GRUMBERG, O., AND LONG, D. 1994. Model checking and abstraction. *Trans. on Prog. Lang. and Syst. 16*, 5 (Sept.), 1512–1542.

CLARKE, E., GRUMBERG, O., AND PELED, D. 1999a. *Model Checking*. MIT Press.

CLARKE, E., TALUPUR, M., AND AND, H. V. 2008. Proving ptolemy right: The environment abstraction framework for model checking concurrent systems. In *Tools and Algorithms for the Construction and Analysis of Systems*. Lecture Notes in Computer Science, vol. 4963. Springer, 33–47.

CLARKE, E., TALUPUR, M., AND VEITH, H. 2006. Environment abstraction for parameterized verification. In *Verification, Model Checking, and Abstract Interpretation*. Lecture Notes in Computer Science, vol. 3855. Springer, 126–141.

CLARKE, E. M., GRUMBERG, O., AND JHA, S. 1997. Verifying parameterized networks. *Trans. on Prog. Lang. and Syst. 19*, 5 (Sept.), 726–750.

CLARKE, E. M., GRUMBERG, O., AND PELED, D. A. 1999b. *Model Checking*. The MIT Press.

CLAVEL, M., DURÁN, F., EKER, S., LINCOLN, P., MARTÍ-OLIET, N., MESEGUER, J., AND QUESADA, J. F. 2002. Maude: specification and programming in rewriting logic. *Theor. Comput. Sci. 285*, 2, 187–243.

COOK, B., KROENING, D., AND SHARYGINA, N. 2005. Symbolic model checking for asynchronous boolean programs. In *Model Checking Software (SPIN)*. LNCS. Springer, 75–90.

CORBETT, J. C. 2000. Using shape analysis to reduce finite-state models of concurrent java programs. *ACM Trans. Softw. Eng. Methodol. 9*, 1, 51–93.

COUSOT, P. AND COUSOT, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Proc. Symp. on Principles of Prog. Languages*. ACM Press, New York, NY, 238–252.

COUSOT, P. AND COUSOT, R. 1979. Systematic design of program analysis frameworks. In *Proc. Symp. on Principles of Prog. Languages*. ACM Press, New York, NY, 269–282.

DAS, M., LERNER, S., AND SEIGLE, M. 2002. Esp: path-sensitive program verification in polynomial time. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 57–68.

DAS, S., DILL, D., AND PARK, S. 1999. Experience with predicate abstraction. In *11th Int. Conf. on Computer-Aided Verification*. Springer-Verlag. Trento, Italy.

DEMARTINI, C., IOSIF, R., AND SISTO, R. 1999a. A deadlock detection tool for concurrent Java programs. *Software: Practice and Experience 29*, 7 (June), 577–603.

DEMARTINI, C., IOSIF, R., AND SISTO, R. 1999b. dSPIN : A dynamic extension of SPIN.

DISTEFANO, D., O'HEARN, P. W., AND YANG, H. 2006. A local shape analysis based on separation logic. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. LNCS, vol. 3920. Springer, 287–302.

EMERSON, E. A. AND SISTLA, A. P. 1993. Symmetry and model checking. In *CAV*, C. Courcoubetis, Ed. Lecture Notes in Computer Science, vol. 697. Springer, 463–478.

ENGLER, D. AND ASHCRAFT, K. 2003. Racerx: effective, static detection of race conditions and deadlocks. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. ACM Press, New York, NY, USA, 237–252.

FARZAN, A., CHEN, F., MESEGUER, J., AND ROSU, G. 2004. Formal analysis of java programs in javafan. In *CAV*, R. Alur and D. Peled, Eds. Lecture Notes in Computer Science, vol. 3114. Springer, 501–505.

FINK, S., YAHAV, E., DOR, N., RAMALINGAM, G., AND GEAY, E. 2006. Effective typestate verification in the presence of aliasing. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*. ACM Press, New York, NY, USA, 133–144.

FLANAGAN, C. AND ABADI, M. 1999. Types for safe locking. In *ESOP '99: Proceedings of the 8th European Symposium on Programming Languages and Systems*. Springer-Verlag, London, UK, 91–108.

FLANAGAN, C. AND FREUND, S. N. 2000. Type-based race detection for java. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 219–232.

FLANAGAN, C. AND FREUND, S. N. 2004. Type inference against races. In *SAS*, R. Giacobazzi, Ed. Lecture Notes in Computer Science, vol. 3148. Springer, 116–132.

FLANAGAN, C., FREUND, S. N., AND QADEER, S. 2002. Thread-modular verification for shared-memory programs. In *11th European Symposium on Programming, ESOP 2002*. Lecture Notes in Computer Science, vol. 2305. Springer, 262–277.

FLANAGAN, C., FREUND, S. N., AND YI, J. 2008. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*. ACM, New York, NY, USA, 293–303.

FLANAGAN, C. AND GODEFROID, P. 2005. Dynamic partial-order reduction for model checking software. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, New York, NY, USA, 110–121.

GANAI, M. K. AND GUPTA, A. 2008. Efficient modeling of concurrent systems in BMC. In *Model Checking Software (SPIN)*. LNCS, vol. 5156. Springer, 114–133.

GODEFROID, P. 1996. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*. Lecture Notes in Computer Science, vol. 1032. Springer-Verlag.

GOETZ, B., PEIERLS, T., BLOCH, J., BOWBEER, J., HOLMES, D., AND LEA, D. 2006. *Java Concurrency in Practice*. Addison-Wesley Professional.

GOPAN, D., REPS, T., AND SAGIV, M. 2005. A framework for numeric analysis of array operations. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, New York, NY, USA, 338–350.

GOSLING, J., JOY, B., AND STEELE, G. 1997. *The Java Language Specification*. The Java Series. Addison-Wesley.

GOTSMAN, A., BERDINE, J., COOK, B., RINETZKY, N., AND SAGIV, M. 2007. Local reasoning for storable locks and threads. In *Programming Languages and Systems, 5th Asian Symposium, APLAS 2007*. Lecture Notes in Computer Science, vol. 4807. Springer, 19–37.

GOTSMAN, A., BERDINE, J., COOK, B., AND SAGIV, M. 2007. Thread-modular shape analysis. In *PLDI '07: Proceedings of the 2007 conference on Programming language design and implementation*. ACM, New York, NY, USA, 266–277.

GOTSMAN, A., COOK, B., PARKINSON, M., AND VAFEIADIS, V. 2009. Proving that non-blocking algorithms don't block. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, New York, NY, USA, 16–28.

GRAF, S. AND SAIDI, H. 1997. Construction of abstract state graphs with PVS. *LNCS 1254*, 72–83.

GUETA, G., FLANAGAN, C., YAHAV, E., AND SAGIV, M. 2006. A semantics for reducing nondeterminism in concurrent programs and its applications. Technical Report TA-CS-2006-052, School of Computer Science, Tel Aviv University. Avialable at "http://www.cs.tau.ac.il/∼msagiv/skipping.ps".

GUETA, G., FLANAGAN, C., YAHAV, E., AND SAGIV, M. 2007. Cartesian partial order reduction. In *SPIN '07: Proceedings of the 14th Workshop on Model Checking Software*. Lecture Notes in Computer Science. Springer.

HAVELUND, K. AND PRESSBURGER, T. 2000. Model checking Java programs using Java pathfinder. *Int. J.on Soft. Tools for Technology Transfer 2*, 4 (Apr.), 366–381.

HENZINGER, T. A., JHALA, R., AND MAJUMDAR, R. 2004. Race checking by context inference. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*. ACM, New York, NY, USA, 1–13.

HERLIHY, M. 1991. Wait-free synchronization. *ACM Transactions of Programming Languages and Systems 13*, 1, 124–149.

HERLIHY, M. AND SHAVIT, N. 2008. *The Art of Multiprocessor Programming*. Morgan and Kaufmann, Burlington.

HERLIHY, M. P. AND WING, J. M. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*, 3, 463–492.

HOLZMANN, G. J. 1995. Proving properties of concurrent systems with SPIN. In *Proc. of the 6th Int. Conf. on Concurrency Theory (CONCUR'95)*. LNCS, vol. 962. Springer, Berlin, GER, 453–455.

ibm 1983. *IBM System/370 Extended Architecture, Principles of Operation*. Publication No. SA22-7085.

IMMERMAN, N. 1998. *Descriptive Complexity*. Texts in Computer Science. Springer-Verlag.

ISHTIAQ, S. S. AND O'HEARN, P. W. 2001. Bi as an assertion language for mutable data structures. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, New York, NY, USA, 14–26.

JEANNET, B., LOGINOV, A., REPS, T. W., AND SAGIV, S. 2004. A relational approach to interprocedural shape analysis. In *Static Analysis, 11th International Symposium, SAS 2004*. Lecture Notes in Computer Science, vol. 3148. Springer, 246–264.

JONES, N. AND MUCHNICK, S. 1981. Flow analysis and optimization of Lisp-like structures. In *Program Flow Analysis: Theory and Applications*, S. Muchnick and N. Jones, Eds. Prentice-Hall, Englewood Cliffs, NJ, Chapter 4, 102–131.

JONES, N. AND MUCHNICK, S. 1982. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Proc. Symp. on Principles of Prog. Languages*. ACM Press, New York, NY, 66–74.

KNAPP, A., CENCIARELLI, P., REUS, B., AND WIRSING, M. 1998. An event-based structural operational semantics of multi-threaded java.

LAHIRI, S. AND QADEER, S. 2008. Back to the future: revisiting precise program verification using smt solvers. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, New York, NY, USA, 171–182.

LAHIRI, S. K., QADEER, S., AND RAKAMARI, Z. 2009. Static and precise detection of concurrency errors in systems code using SMT solvers. In *Computer Aided Verification (CAV)*. LNCS, vol. 5643. Springer, 509–524.

LAL, A. AND REPS, T. 2008. Reducing concurrent analysis under a context bound to sequential analysis. In *Computer Aided Verification*. Lecture Notes in Computer Science, vol. 5123. Springer, 37–51.

LEA, D. 1997. *Concurrent Programming in Java*. Addison-Wesley, Reading, Massachusetts.

LEINO, K. R. AND MÜLLER, P. 2009. A basis for verifying multi-threaded programs. In *ESOP '09: Proceedings of the 18th European Symposium on Programming Languages and Systems*. Springer-Verlag, Berlin, Heidelberg, 378–393.

LEV-AMI, T. AND SAGIV, M. 2000. TVLA: A framework for Kleene based static analysis. In *Proc. Static Analysis Symp.* LNCS, vol. 1824. Springer-Verlag, 280–301.

LINDHOLM, T. AND YELLIN, F. 1997. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA.

MANEVICH, R., LEV-AMI, T., SAGIV, M., RAMALINGAM, G., AND BERDINE, J. 2008. Heap decomposition for concurrent shape analysis. In *Static Analysis, 15th International Symposium, SAS 2008*. Lecture Notes in Computer Science, vol. 5079. Springer, 363–377.

MANEVICH, R., YAHAV, E., RAMALINGAM, G., AND SAGIV, S. 2005. Predicate abstraction and canonical abstraction for singly-linked lists. In *VMCAI '05: 6th International Conference on Verification, Model Checking, and Abstract Interpretation*. 181–198.

MICHAEL, M. AND SCOTT, M. 1996. 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)*. ACM, New York, USA, 267–275.

MICHAEL, M. M. 2004. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems 15*, 6, 491–504.

MOLLER, A. AND SCHWARTZBACH, M. I. 2001. The pointer assertion logic engine. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*. ACM, New York, NY, USA, 221–231.

MOORE, J. S. AND PORTER, G. 2002. The apprentice challenge. *ACM Transactions on Programming Languages and Systems (TOPLAS) 24*, 3, 193–216.

NAIK, M., AIKEN, A., AND WHALEY, J. 2006. Effective static race detection for java. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 308–319.

NETZER, R. AND MILLER, B. 1992. What are race conditions? some issues and formalizations. *ACM Letters on Programming Languages and Systems 1*, 1 (Mar.), 74–88.

NIELSON, F., NIELSON, H. R., AND SAGIV, M. 2000. A kleene analysis of mobile ambients. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*. Springer-Verlag, London, UK, 305–319.

O'HEARN, P. W. 2007. Resources, concurrency, and local reasoning. *Theor. Comput. Sci. 375*, 1-3, 271–307. Prelim version appeared in CONCUR'04, LNCS 3170, pp49-67.

PNUELI, A., XU, J., AND ZUCK, L. D. 2002. Liveness with (0, 1, infty)-counter abstraction. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*. Springer-Verlag, London, UK, 107–122.

PRAKASH, S., LEE, Y., AND JOHNSON, T. 1991. A non-blocking algorithm for shared queues using Compare-and-Swap. In *Proceedings of the 1991 International Conference on Parallel Processing*. 68–75.

PRATIKAKIS, P., FOSTER, J., AND HICKS, M. 2006. Locksmith: context-sensitive correlation analysis for race detection. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 320–331.

QADEER, S. AND REHOF, J. 2005. Context-bounded model checking of concurrent software. In *Tools and Algorithms for the Construction and Analysis of Systems*. Lecture Notes in Computer Science, vol. 3440. Springer, 93–107.

REPS, T., SAGIV, M., AND LOGINOV, A. 2003. Finite differencing of logical formulas for static analysis. In *In Proc. European Symp. on Programming*. LNCS, vol. 2618. Springer-Verlag.

REPS, T. W., SAGIV, M., AND WILHELM, R. 2004. Static program analysis via 3-valued logic. In *Computer Aided Verification*. Lecture Notes in Computer Science, vol. 3114. Springer, 401–404.

REYNOLDS, J. C. 2002. Separation logic: A logic for shared mutable data structures. *Logic in Computer Science, Symposium on 0*, 55.

RINETSKEY, N. AND SAGIV, M. 2001. Interprocedural shape analysis for recursive programs. In *Proc. Intl. Conf. on Compiler Construction*, R. Wilhelm, Ed. LNCS, vol. 2027. Springer-Verlag, 133–149.

RINETZKY, N. 2008. Interprocedural and modular local heap shape analysis. Ph.D. thesis, Tel Aviv University.

SAGIV, M., REPS, T., AND WILHELM, R. 1998. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Program. Lang. Syst. 20*, 1 (Jan.), 1–50.

SAGIV, M., REPS, T., AND WILHELM, R. 2002. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems 24*, 3, 217–298.

SAIDI, H. 2000. Model checking guided abstraction and analysis. In *SAS '00: Proceedings of the 7th International Symposium on Static Analysis*. Springer-Verlag, London, UK, 377–396.

SHAHAM, R., YAHAV, E., KOLODNER, E. K., AND SAGIV, M. 2003. Establishing local temporal heap safety properties with applications to compile-time memory management. In *Proc. of the 10th International Static Analysis Symposium, SAS 2003*. LNCS, vol. 2694. Springer-Verlag.

SILBERSCHATZ, A. AND GALVIN, P. B. 1994. *Operating Systems Concepts*, 4 ed. Addison-Wesley, Reading.

STERLING, N. 1993. Warlock — a static data race analysis tool. In *USENIX Technical Conference Proceedings*. 97–106.

STOLLER, S. 2000. Model-checking multi-threaded distributed Java programs. In *Proc. 7th Int. SPIN Workshop on Model Checking of Software*. LNCS, vol. 1885. Springer-Verlag, 224–244.

STONE, J. M. 1990. A simple and correct shared-queue algorithm using compare-and-swap. In *Supercomputing '90: Proceedings of the 1990 conference on Supercomputing*. IEEE Computer Society Press, Los Alamitos, CA, USA, 495–504.

STONE, J. M. 1992. A non-blocking Compare-and-Swap algorithm for a shared circular queue. In *Parallel and Distributed Computing in Engineering Systems*, S. Tzafestas et al., Eds. Elsevier Science Publishers, 147–152.

STROM, R. AND YEMINI, S. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng. 12*, 1, 157–171.

STROM, R. E. 1983. Mechanisms for compile-time enforcement of security. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM Press, New York, NY, USA, 276–284.

VAFEIADIS, V. 2008. Modular fine-grained concurrency verification. Ph.D. thesis, University of Cambridge, Computer Laboratory. Also available as technical report UCAM-CL-TR-726.

VAFEIADIS, V. 2009. Shape-value abstraction for verifying linearizability. In *Verification, Model Checking, and Abstract Interpretation*. Lecture Notes in Computer Science, vol. 5403. Springer, 335–348.

VALLÉE-RAI, R., HENDREN, L., SUNDARESAN, V., LAM, P., GAGNON, E., AND CO, P. 1999. Soot - a java optimization framework. In *Proc. of CASCON 1999*. 125–135.

VALMARI, A. 1991. Stubborn sets for reduced state space generation. In *Advances in Petri Nets 1990*. Lecture Notes in Computer Science, vol. 483. Springer-Verlag, 491–515.

VECHEV, M. AND YAHAV, E. 2008. Deriving linearizable fine-grained concurrent objects. In *PLDI '08: Proceedings of the 2008 conference on Programming language design and implementation*. ACM, 125–135.

VECHEV, M., YAHAV, E., BACON, D. F., AND RINETZKY, N. 2007. Cgcexplorer: a semi-automated search procedure for provably correct concurrent collectors. In *PLDI '07: Proceedings of the 2007 conference on Programming language design and implementation*. ACM, New York, NY, USA, 456–467.

VECHEV, M., YAHAV, E., AND YORSH, G. 2009. Experience with model checking linearizability. In *Proceedings of the 16th International SPIN Workshop on Model Checking Software*. Springer, 261–278.

VERMEULEN, A. 1997. Java deadlock: The woes of multithreaded design. *Dr. Dobb's Journal of Software Tools 22*, 9 (Sept.), 52, 54–56, 88, 89.

WING, J. M. AND GONG, C. 1990. A library of concurrent objects and their proofs of correctness. Tech. Rep. CMU–CS–90–151, CMU.

YAHAV, E. 2000. 3VMC user's manual. Available at http://www.math.tau.ac.il/∼yahave.

YAHAV, E. 2001. Verifying safety properties of concurrent Java programs using 3-valued logic. In *Proc. Symp. on Principles of Prog. Languages*. 27–40.

YAHAV, E. AND RAMALINGAM, G. 2004. Verifying safety properties using separation and heterogeneous abstractions. In *Proceedings of the 2004 conference on Programming language design and implementation*. ACM Press, 25–34.

YAHAV, E. AND SAGIV, M. 2003. Automatically verifying concurrent queue algorithms. In *Electronic Notes in Theoretical Computer Science*, B. Cook, S. Stoller, and W. Visser, Eds. Vol. 89. Elsevier.

YANG, H., LEE, O., BERDINE, J., CALCAGNO, C., COOK, B., DISTEFANO, D., AND OHEARN, P. 2008. Scalable shape analysis for systems code. In *Computer Aided Verification*. Lecture Notes in Computer Science, vol. 5123. Springer, 385–398.

ZEE, K., KUNCAK, V., AND RINARD, M. 2008. Full functional verification of linked data structures. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*. ACM, New York, NY, USA, 349–361.

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. *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* $(V \setminus \{v_1, v_2\}) \cup \{v_3, v_4\}$.

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. *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 $[\![\varphi]\!]_2^S(Z) = 1$. 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 to define the meaning of a formula with respect to a 3-valued structure.

DEFINITION A.3. *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.*

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 $S, Z \models_3 \varphi$.*