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

Eran Yahav*

Abstract

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

- Concurrent manipulation of linked-list based ADT preserves the ADT datatype invariant [18].
- 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 find bugs in erroneous versions of such implementations.

A prototype of our framework has been implemented.

1 Introduction

Java provides low-level concurrency-control constructs that enable the programmer to create complicated and powerful synchronization schemes. The Java language provides no means of compile-time or run-time checking for the correctness of concurrent behavior. This makes concurrent programming in Java quite error-prone (e.g., [38]).

The theme of this paper is to develop compile-time techniques for verifying safety properties by detecting program configurations that may violate desired properties.

*Computer Sciences Department, School of Mathematical Sciences, Tel-Aviv University, Tel-Aviv 69978, Tel: +972-3-6407606, Fax: +972-3-6406761, yahave@math.tau.ac.il.

1.1 Main Results and Related Work

In this paper, we present a framework for verifying safety properties of concurrent Java programs. This framework handles dynamic allocation of objects and references to objects. This allows us to analyze programs that dynamically allocate thread objects, and even programs that create an unbounded number of threads. Dynamic allocation of threads is common when implementing services in threads (e.g., [22], ch. 6). For these programs, we can verify properties such as the absence of interference. Handling dynamically allocated objects also allows us to model concurrent programs that manipulate linked-lists in the most precise known way.

1.1.1 A Parametric Framework for Verifying Safety Properties

We provide a parametric framework for verifying safety properties of concurrent Java programs. We use different instances of this framework (see Section 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., [21]) 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 verify that a safety property holds in a given configuration. Our framework then computes a safe approximation to 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 [11]. 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 is violated (*false alarm*) but can never miss a violation.

Our framework can be viewed as on-the-fly model-checking [8] for verifying safety properties of programs. On-the-fly model checking does not require a construction of a global state graph as a prerequisite for property verification. In order to handle dynamic creation and references to objects, we use first-order logical structures to represent

configurations of the program. A *state-space exploration* algorithm (see Figure 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 as *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 ([7]). This could be trivially encoded in our framework by using only nullary predicates (and thus the number of individuals in a logical structure is immaterial). In fact, our framework allows more general (and natural) modeling of programs by using unary and binary predicates. This is crucial in order to handle dynamically allocated objects and references to objects where the "name" of the object is unknown at compile-time. Even the technique of [15] (formulated for processes rather than threads) relies on explicit process names, and thus cannot handle dynamic allocation of processes.

JavaPathFinder [17] and Java2Spin [13] translate Java source code to PROMELA representation. The SPIN model-checker [19] 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 [14] supports dynamic allocation of objects. However, since it uses no abstraction, it can only handle bounded data-structures and bounded number of threads.

[6] presents a method for the verification of parametric families of systems. A network grammar is used to construct a process invariant that simulates all systems in the family. However, it can not handle dynamic allocation of objects.

Recent work [34] presents new abstraction predicates but does not have the notion of summary nodes. Thus, it can not handle programs with unbounded number of allocated objects. Moreover, our framework presents a model checking algorithm that recognizes abstraction as suggested there.

In our framework, rather than having a separate model-extraction and model-checking phases, we follow the abstract-interpretation approach [11] and cast our analysis in a syntax-directed manner.

Technically speaking, our framework is a generalization of [33] in the following aspects: (i) Program configurations are used to model the global state of the program instead of modeling only the relationships between heap-allocated objects. This allows us to combine thread scheduling information with information about the shape of the heap. (ii) Program control-flow is not separately represented, but instead the program location of each thread is maintained in the configuration which allows us to handle an unbounded number of threads in a natural way. This is naturally coded in first-order logic as a property of a thread (in contrast to model checking in which it is externally coded). Furthermore, it does not require control-flow information to be computed in a separate earlier phase. This is an advantage because the imprecision in control-flow computation could lead to imprecise results. (iii) We use the standard interleaving model of concurrency. A slightly different generalization is used in [30], which even allows the program to modify itself to support the semantics of Mobile Ambients [4].

The FLAVERS system [28] uses trace flow graphs with feasibility constraints represented as finite-state automata to model the semantics of concurrent Java programs.

An important difference between our framework and FLAVERS is that our framework has the ability to model the dynamic creation of objects and threads. It should be noted that FLAVERS can only detect violation of properties represented as finite state automata in terms of observable events in the program, and thus can not detect some of the properties verified by our framework (e.g., deadlock). Moreover, since every finite automaton can be trivially coded in our framework, it generalizes FLAVERS. However, the cost of doing that in our current implementation is higher.

In [36], a framework for model checking distributed Java programs is presented. This framework uses partial-order methods to reduce the size of the explored state-space. However, it uses no abstraction and thus can only handle bounded data-structures and bounded number of threads. We intend to use similar partial-order methods in future versions of our framework.

Bandera [10] is a framework for translating Java programs to a program model acceptable by existing model-checking tools. During translation, the model is reduced using slicing and other program analyses. Bandera imposes a bound on the number of allocated objects and does not allow dynamic removal of objects.

In [9], shape analysis of concurrent programs is used to reduce finite-state models of concurrent Java programs. In this analysis, the number of threads is bounded. The algorithm presented is based on [5], which uses a single *shape graph* for each program location, and uses an abstraction which leads to overly imprecise results (e.g., in programs that traverse data structures based on allocation sites).

In recent work [37], shape analysis of concurrent programs is used for eliminating synchronization. As in [9], the algorithm presented is an extension of [5] and suffers from the same imprecision. It should be noted that despite the different goals of our work, it is significantly more precise. In particular, it always performs strong updates.

In [1], static analysis is used to identify opportunities to eliminate unnecessary synchronization. That work assumes a static control-flow-graph, and ignores thread-scheduling mechanisms.

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 [29]). 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 [26], also shown in the appendix) we are able to show that write-write interference is not possible since writing is never performed on the same object.

Deadlock: Our framework has been used to verify the absence of a few types of deadlocks: (i) total deadlocks in which all threads are blocked. (ii) nested monitors deadlocks, which are very common in Java ([38]) (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 [22], 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 a scheduling queue can be used to precisely reason about thread scheduling.

For example, Figure 1 shows a concurrent program using a queue. The implementation of the queue is given in Figure 2 and Figure 3. This program is used as a running example throughout this paper. 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 ill-synchronized version of the same queue (not shown here).

Illegal Thread Interactions: The Java semantics allows the programmer to introduce thread interactions that are illegal and result in an exception during program execution. For example - starting a thread more than once will result with an `IllegalThreadStateException` being thrown. Our framework has been used to detect such illegal interactions.

1.1.3 Prototype Implementation

We have implemented a prototype of our framework using TVLA [23]. We applied the analyses to several small but interesting programs. The results are reported in Section 6.

Currently, we do not perform interprocedural analysis and assume that procedures are inlined. However, our framework does support recursive procedures based on [31]. The main disadvantage of our current implementation is that no optimization is used, and thus only small programs can be handled.

We are encouraged by the precision of our results and the simplicity of the implementation.

1.1.4 Outline of the Paper

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. Finally, Section 7 concludes the paper and discusses further work.

2 Java Concurrency Model

We now give a short description of the Java concurrency-primitives used in this paper. The reader is referred to [16, 22, 24] for more details.

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

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

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

Figure 1: A simple program that uses a queue.

- 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 assume that threads are scheduled arbitrarily.

The program shown in Figure 1 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

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

Figure 2: Simplified Java source code for a queue implementation.

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

Figure 3: Simplified Java source code for a QueueItem implementation.

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

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

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

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

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

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

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

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.

Due to space limitations we do not present the semantics for multiple acquisitions of a lock by the same thread. These details can be found in [39].

3 A Program Model

In this section we lay the ground to our analysis of Java programs. 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.

Predicates	Intended Meaning
$is_thread(t)$	t is a thread
$\{at[lab](t) : lab \in Labels\}$	thread t is at label lab
$\{rvalue[fld](o_1, o_2) : fld \in Fields\}$	field fld of the object o_1 points to the object o_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
$idlt(l_1, l_2)$	the id of lock l_1 is less than the id of lock l_2

Table 1: Predicates for partial Java semantics.

3.1 Representing Program Configurations via Logical Structures

A *program configuration* encodes a global state of a program which consists of (i) a global store, (ii) the program-location of every thread, and (iii) the status of locks and threads, e.g., if a thread is waiting for a lock. Technically, first-order logic with transitive-closure is used in this 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 1 contains the predicates used to analyze our running example program.

- The unary predicate $is_thread(t)$ is used to denote the objects that are threads, i.e., instances of a subclass of `class Thread`.
- For every potential program-location (program label) lab of a thread t, there is a unary predicate $at[lab](t)$ which is true when t is at lab.
- For every class field and function parameter `fld`, a binary predicate $rvalue[fld](o_1, o_2)$ records the fact that the `fld` of the object o_1 points to the object o_2.
- The predicates $held_by(l, t)$, $blocked(t, l)$ and $waiting(t, l)$ model possible relationships between locks and threads.
- The predicate $idlt(l_1, l_2)$ is used to record a partial order between objects. Each object is assumed to have a unique id. The predicate $idlt(l_1, l_2)$ 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 [35]. In Section 5.1 we show how our technique can be used to verify that a program uses such a resource-allocation policy.

Note that predicates in Table 1 are actually written in a generic way and can be applied to analyze different Java programs by modifying the set of labels and fields.

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

- $U^\natural$ is the universe of the 2-valued structure. Each individual in $U^\natural$ represents an allocated heap object.
- $\iota^\natural$ is the interpretation function mapping predicates to their truth-value in the structure, i.e., for every predicate $p \in P$ of arity k, $\iota^\natural(p) : U^{\natural k} \to \{0, 1\}$.

Configurations are depicted as directed graphs. Each individual from the universe is displayed as a node. A unary predicate p which holds for an individual (node) u is drawn inside the node u. The name of a node is written inside angle brackets. Node names are only used for ease of presentation and do not affect the analysis. A true binary predicate $p(u_1, u_2)$ is drawn as directed edge from u_1 to u_2 labeled with the predicate symbol. 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 Figure 4 corresponds to a global state of the example program with 5 threads: a single *producer* thread (labeled prd) which acquired the queue's lock, a single *consumer* thread (labeled cns) which is blocked on the queue's lock, and 3 *approving* threads ($a1, a2, a3$) which haven't performed any action yet. The role of the predicate $r_by[fld](o)$ will be explained in future sections. For simplicity 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 $rvalue[next](o_1, o_2)$ records for each object o_1 the target object referenced by its `next` field.

Note that the universe $U^\natural$ is not bounded since the analyzed program may allocate new non-thread and/or thread individuals. We do not place a bound on 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 configurations. For example, the formula

$$\exists t : is_thread(t) \wedge held_by(l, t) \tag{1}$$

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 logics.

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:

$$\forall t_1 \forall t_2 : (t_1 \neq t_2) \to \neg(at[l_{crit}](t_1) \wedge at[l_{crit}](t_2)) \tag{2}$$

3.3 A Structural Operational Semantics of Configurations

Figure 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

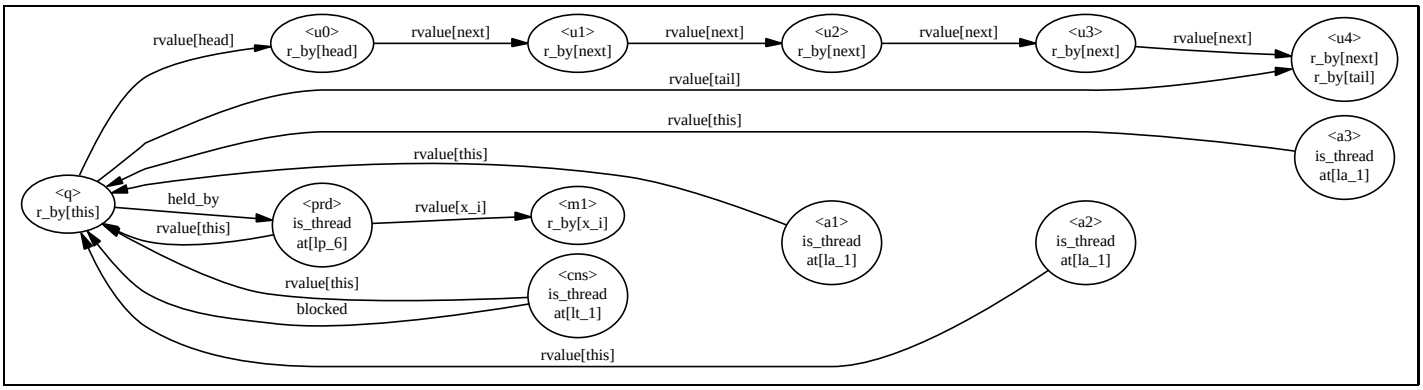

Figure 4: A concrete configuration $C_4{}^\sharp$.

```
initialize(C_0) {
    for each C ∈ C_0
        push(stack,C)
}
explore() {
  while stack is not empty {
    C = pop(stack)
    if not member(C, stateSpace) {
       verify(C)
       stateSpace' = stateSpace ∪ {C}
       for each action ac
         for each C' such that C ⇒_ac C'
           push(stack,C')
    }
  }
}
```

Figure 5: State space exploration.

essentially the classic state-space exploration used in model-checking [8]. 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 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.
- Enabled actions create a new configuration where the interpretations of every predicate p of arity k is determined by evaluating a formula $\varphi_p(v_1, v_2, \ldots, v_k)$ which may use $v_1, v_2, \ldots, v_k$ and t_s as well as all other predicates in P.

Table 2 defines the semantics of concurrency statements used in the running example. Operational semantics for other statements can be found in the appendix and in [39]. The table lists a precondition and an update formulae for each action. Predicates not appearing in the update formulae are assumed to remain unchanged by the action. The set of actions is partitioned to blocking and non-blocking actions. Blocking actions do not affect the program-location. Non blocking actions advance to the next program-location by updating the $at[lab](t_s)$ predicates for the thread.

A Java statement may be modeled by several alternative actions corresponding to the different behaviors of the statement. A single action to be taken is determined by evaluation of the preconditions. The actions $lock(v)$ and $blockLock(v)$ correspond to the two possible behaviors on entry to a `synchronized(v)` block: $lock(v)$ is enabled when there exists no thread (other than the current thread) that is holding the lock referenced by v, $blockLock(v)$ is enabled when such thread exists. The action $unlock(v)$ corresponds to the release of the lock upon exit of the `synchronized(v)` block. The action $wait(v)$ corresponds to invocation of `v.wait()`. The actions $notify(v)$ and $ignoredNotify(v)$ correspond to the possible behaviours when calling `v.notify()`: $notify(v)$ is enabled when a there exists a thread waiting on the lock referenced by v, $ignoredNotify(v)$ is enabled when no such thread exists. $notifyAll(v)$ and $ignoredNotifyAll(v)$ model similar behavior of `v.notifyAll()`. Technically, the translation of a Java statement (and condition) to several alternative actions can be performed by a front-end.

Formally, the meaning of actions is defined as follows:

Definition 3.2 *We say that $C^\sharp = \langle U, \iota \rangle$ rewrites into a configuration $C^{\sharp\prime} = \langle U, \iota' \rangle$ (denoted by $C^\sharp \Rightarrow_{ac} C^{\sharp\prime}$) where ac is an action, if there exists an assignment Z that satisfies the precondition of ac on $C^\sharp$, and for every $p \in P$ of arity k and $u_1, \ldots, u_k \in U$,*

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

$$[\![\varphi_p(v_1, v_2, \ldots, v_k)]\!]_2^{C^\sharp}(Z[v_1 \mapsto u_1, v_2 \mapsto u_2, \ldots, v_k \mapsto u_k])$$

where $\varphi_p(v_1, \cdots, v_k)$ is the formula for p given in Table 2.

In addition, there is a special action that creates a new individual u_{new} and results with a structure $C^{\sharp\prime} = \langle U \cup \{u_{new}\}, \iota' \rangle$. A special predicate $isNew$ holds for u_{new}, and thus can be used in the predicate update formulae.

Action	Precondition	Predicate-update
$lock(v)$	$\neg\exists t \neq t_s : rvalue[v](t_s, l) \wedge held_by(l, t)$	$held_by'(l_1, t_1) = held_by(l_1, t_1) \vee (t_1 = t_s \wedge l_1 = l)$ $blocked'(t_1, l_1) = blocked(t_1, l_1) \wedge ((t_1 \neq t_s) \vee (l_1 \neq l))$
$unlock(v)$	$rvalue[v](t_s, l)$	$held_by'(l_1, t_1) = held_by(l_1, t_1) \wedge (t_1 \neq t_s \vee l_1 \neq l)$
$wait(v)$	$rvalue[v](t_s, l)$	$held_by'(l_1, t_1) = held_by(l_1, t_1) \wedge (t_1 \neq t_s \vee l_1 \neq l)$ $waiting'(t_1, l_1) = waiting(t_1, l_1) \vee (t_1 = t_s \wedge l_1 = l)$
$notify(v)$	$rvalue[v](t_s, l) \wedge \exists t_w : waiting(t_w, l)$	$waiting'(t_1, l_1) = waiting(t_1, l_1) \wedge (t_1 \neq t_w \vee l_1 \neq l)$ $blocked'(t_1, l_1) = blocked(t_1, l_1) \vee (t_1 = t_w \wedge l_1 = l)$
$ignoredNotify(v)$	$rvalue[v](t_s, l) \wedge \neg\exists t_w : waiting(t_w, l)$	
$notifyAll(v)$	$rvalue[v](t_s, l) \wedge \exists t_w : waiting(t_w, l)$	$waiting'(t_1, l_1) = waiting(t_1, l_1) \wedge (l_1 \neq l)$ $blocked'(t_1, l_1) = blocked(t_1, l_1) \vee (waiting(t_1, l_1) \wedge (l_1 = l))$
$ignoredNotifyAll(v)$	$rvalue[v](t_s, l) \wedge \neg\exists t_w : waiting(t_w, l)$	
$blockLock(v)$	$\exists t \neq t_s : rvalue[v](t_s, l) \wedge held_by(l, t)$	$blocked'(t_1, l_1) = blocked(t_1, l_1) \vee (t_1 = t_s \wedge l_1 = l)$

Table 2: Operational semantics for concurrency statements. Actions above the two horizontal lines are non-blocking, the $blockLock(v)$ action is blocking.

We say that a configuration $C^\natural$ transitively rewrites into a configuration $C^{\natural'}$ *(denoted by $C^\natural \Rightarrow^* C^{\natural'}$) if there exists a (potentially empty) sequence of configurations $C^\natural = {C^\natural}_0, {C^\natural}_1, \ldots, {C^\natural}_n = C^{\natural'}$ such that for each $0 \leq i < n$, ${C^\natural}_i \Rightarrow {C^\natural}_{i+1}$.*

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 $\exists C_i \in C_I : C_i \Rightarrow^* C_r$.

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

Our analysis described in Section 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 3 lists the formulae used to detect configurations that violate a safety property.

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 Figure 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 Figure 2. This is due to the fact that synchronization prevents the consumer thread $< cns >$ from being at label lt_3 when the producer thread $< prd >$ is at label lp_6.

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

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

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

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

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

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

Other errors detected using our technique are listed in [39].

4 An Abstract Program Model

The state-space exploration algorithm of Figure 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.

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* where the value 1/2 is called *indefinite value*. Formally, an

Formula	Intended Meaning
$\exists(t_r, t_w)\exists o:\ is_thread(t_r) \wedge is_thread(t_w) \wedge (t_r \neq t_w)$ $\wedge at[lr](t_r) \wedge at[lw](t_w)$ $\wedge rvalue[x_w](t_w, o) \wedge rvalue[x_r](t_r, o)$	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.
$\exists(t_{w1}, t_{w2}) : \exists o:\ is_thread(t_{w1}) \wedge is_thread(t_{w2}) \wedge (t_{w1} \neq t_{w2})$ $\wedge at[lw_1](t_1) \wedge at[lw_2](t_2)$ $\wedge rvalue[x_{w1}](t_{w1}, o) \wedge rvalue[x_{w2}](t_{w2}, o)$	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.
$\forall t : is_thread(t) \rightarrow \exists l : blocked(t, l)$	Total Deadlock
$\exists t : \exists(l_1, l_2):\ is_thread(t) \wedge blocked(t, l_1) \wedge held_by(l_2, t)$ $\wedge \neg idlt(l_2, l_1)$	Resource Ordering. A thread t is blocked on a lock "smaller" than a lock it is holding.
$\exists(t_w, o_{out}, o_{in}):\ is_thread(t_w) \wedge waiting(t_w, o_{in}) \wedge held_by(o_{out}, t_w)$ $\wedge rvalue[in]^*(o_{out}, o_{in})$ $\wedge \forall o_p : ((o_p \neq o_{out}) \wedge rvalue[in]^*(o_{out}, o_p)$ $\wedge rvalue[in]^*(o_p, o_{in})$ $\rightarrow \neg(\exists(t_1, t_2) : rvalue[in](t_1, o_p) \wedge rvalue[in](t_2, o_p))$	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.
$\exists t : at[l_s](t) \wedge rvalue[v](t, l) \wedge \neg held_by(l, t)$	Missing Ownership. Thread invoking `v.wait()` or `v.notify()` at label l_s when not holding the lock referenced by v.
See Section 5.2	Shared ADT
See Section 5.3	Thread Interactions

Table 3: Violations of safety properties detected in this paper.

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

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

An individual $u \in U$ that represents more than a single object is called a *summary node*.

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.

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

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

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

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

Example 4.1 The abstract configuration C_6 *represents* concrete configuration $C_4{}^\sharp$.

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 may represent many configurations. For example, it may represent any configuration with more than 3 queue items. In a similar fashion, the abstract configuration represents configurations with one or more threads that reside at label la_1.

Note that the RW-Interference condition evaluates to 0 over the abstract configuration C_6.

4.2 An Abstract Semantics

We use the same simple algorithm from Figure 5 for exploration of the abstract state space. Two of the operations used by the algorithm are modified to work for abstract configurations: (i) The membership operator $member(C, stateSpace)$ is modified to check if the configuration C can be represented by one of the configurations in $stateSpace$. This is an optimization for preventing exploration of redundant configurations. (ii) 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 C_0 being the abstraction of initial configurations.

Implementing an algorithm for computing the *rewrite* relation on abstract configurations is non-trivial because one

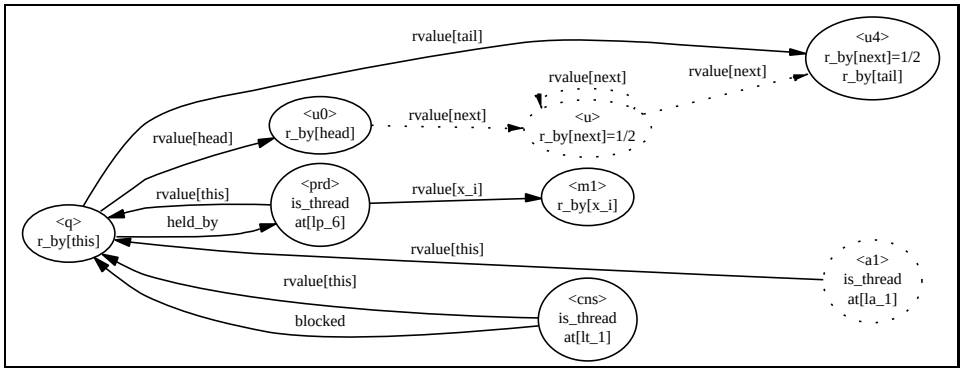

Figure 6: An abstract configuration C_6 *representing* the configuration $C_4{}^\natural$ shown in Figure 4.

has to consider all possible relations on the set of represented (concrete) configurations.

The *best* conservative effect of an action (also known as the *induced* effect of an action) [12] 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 which results with a set of abstract configurations *representing* the results of the action.

Our prototype implementation described in Section 6 operates directly on abstract configurations thereby obtaining 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 3 without producing *false alarms* on our example programs.

Definition 4.2 *We say that an abstract configuration C* rewrites into an abstract configuration *C' (denoted by $C \Rightarrow_{ac} C'$) where ac is an action, if for C and for C' there exists $C^\natural$ and $C^{\natural\prime} = \langle U^\natural, \iota^{\natural\prime}\rangle$ such that: (i) $C^\natural$ is in the concretization of C, i.e., C* represents *$C^\natural$, (ii) C' is the* canonical abstraction *of $C^{\natural\prime}$, (iii) an assignment Z that satisfies the precondition of ac on $C^\natural$, and for every $p \in P$ of arity k and $u_1, \ldots, u_k \in U^\natural$,*

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

where $\varphi_p(v_1, \cdots, v_k)$ is the formula for p given in Table 2. We write $C \Rightarrow C'$ if for some action ac $C \Rightarrow_{ac} C'$.

Example 4.3 The abstract configuration $C_{7,0}$ shown in Figure 7 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 $\{C_{7,0,1}, C_{7,0,2}, \ldots\}$ 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 $\{C_{7,1,1}, C_{7,1,2}, \ldots\}$ is the set after the application of $lock(v)$. The set of abstract configurations $\{C_{7,2,1}, C_{7,2,2}\}$ is the finite set of configurations after abstraction.

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 [33].

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

Example 4.4 Consider an unbounded number of threads competing to acquire a shared resource l. Assume that a thread t_1 already acquired the lock. The configuration $C_{8,0,1}$ shown in Figure 8 corresponds to a state in which some thread tried to acquire the lock l and became blocked on the lock as a consequence. In this configuration, the formula $\exists t : blocked(t, l)$ evaluates to 1/2. Configuration $C_{8,0,2}$ 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 evaluation of the original formula over the configuration with instrumentation also evaluates to 1 rather than to 1/2.

5 Verifying Safety Properties

We use the instrumentation predicates listed in Table 4 to improve the precision of our analyses. The following sections list a more precise formulation of the formulae of Table 3 by 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

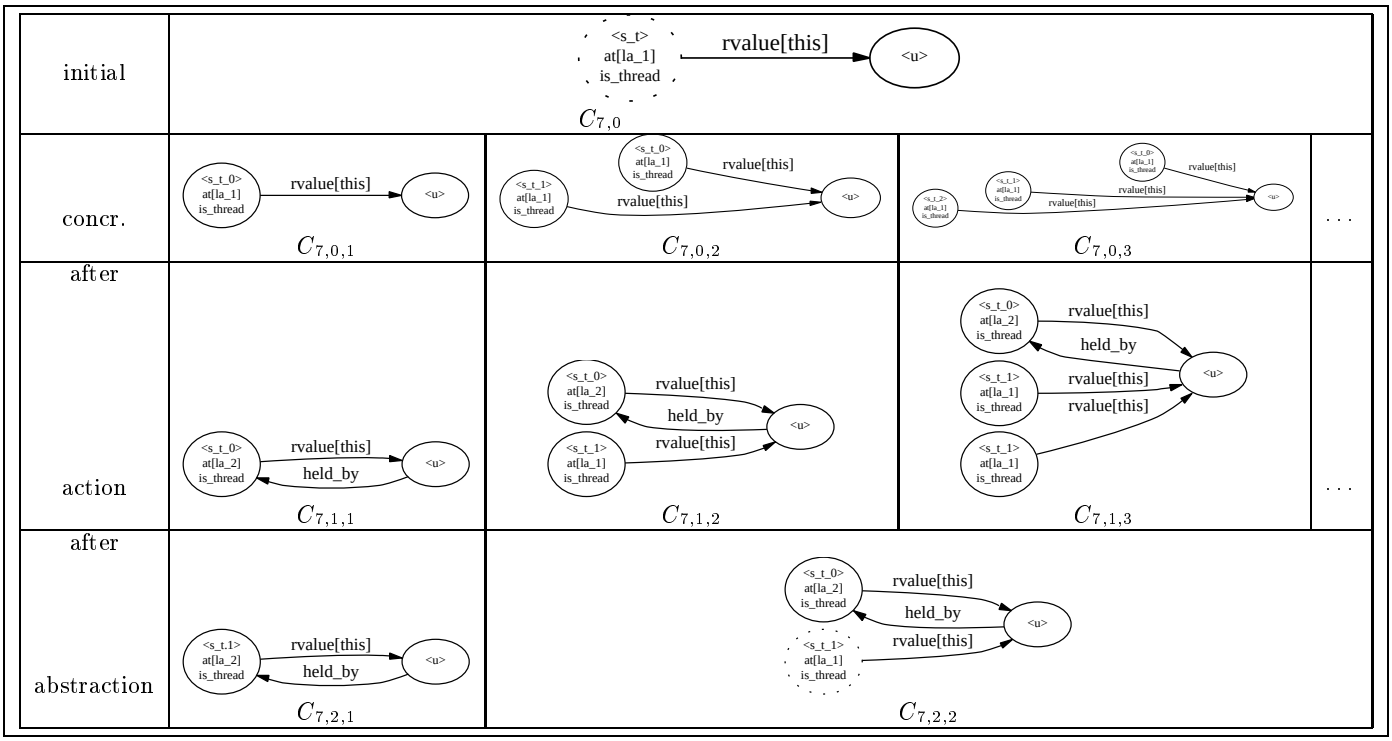

Figure 7: Concretization and predicate-update for an unbounded number of threads all performing the $approveHead()$ method of the running example.

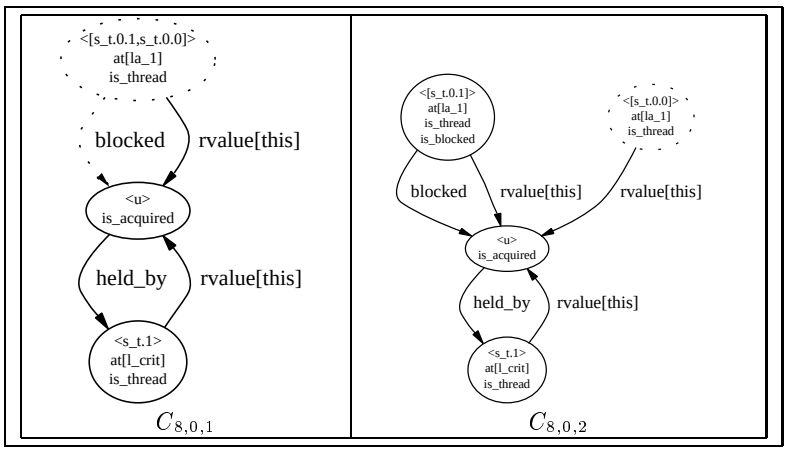

Figure 8: Instrumentation predicate $is_blocked(t)$.

$\exists t : slock(t)$. The formula for nested-monitors deadlock is given below:

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

5.2 Shared Abstract Data Types

We define a set of reachability predicates similar to the ones defined in [33]. 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 no longer reachable from the head of the queue.

Predicate	Intended Meaning	Defining Formula
$is[fld](l_1)$	l_1 is referenced by the field fld of more than one object	$\exists(t_1,t_2):(t_1 \neq t_2) \rightarrow rvalue[fld](t_1,l_1) \wedge rvalue[fld](t_2,l_2)$
$r_by[fld](l)$	l is referenced by the field fld of some object	$\exists o : rvalue[fld](o,l)$
$is_acquired(l)$	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	$\exists(l_1,l_2) : is_thread(t) \wedge blocked(t,l_1) \wedge held_by(l_2,t) \wedge \neg idlt(l_2,l_1)$
$wait_for(t_1,t_2)$	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	$rvalue[fld]^*(o_1,o_2)$
$rt[ref,fld](t,o)$	object o is reachable from thread t by a path starting with a single ref edge followed by any number of fld edges	$\exists o_t : rvalue[v](t,o_t) \wedge rvalue[next]^*(o_t,o)$

Table 4: Instrumentation predicates for partial Java semantics.

5.3 Thread State Errors

We use instrumentation predicates to record thread-state information: $ts_created(t)$, $ts_running(t)$, $ts_blocked(t)$, $ts_waiting(t)$ and $ts_dead(t)$. In order to identify thread-state errors, we add preconditions identifying when an action is illegal or suspicious, these preconditions are listed in Table 5.

Example 5.1 Assume an erroneous version of the running example (Figure 2) in which an unsynchronized version of `put()` is used. Configuration C_9 shown in Figure 9 demonstrates a possible interference in the program identified by our analysis. In the configuration C_9 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 3 evaluates to 1 for this configuration since both threads reference the same object $< u0 >$. Thus RW-Interference is detected.

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

5.4 Unbounded Number of Threads

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

In model checking, the global state of a system is usually described as a tuple containing thread program-counters, and value assignments for shared variables [15]. In [15], symmetry is found between process indices. In our framework, thread names are *canonic names*, that is - only determined by thread properties. Thus, there is no need to explicitly define permutation-equivalence. *The mapping to the canonic names eliminates symmetry in the state space.*

We demonstrate the power of our abstraction by taking the example of a critical section from [15], and verifying that the *mutual exclusion* property holds for an *unbounded number of threads.*

Example 5.2 Consider the *approveHead*() method of class Queue. We would like to verify mutual exclusion over the critical section protected by `sycnronized(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. Figure 10 shows three important abstract configurations arising in the analysis of the example.

6 Prototype Implementation

We have implemented a prototype of our framework using TVLA [23]. We applied the analyses to several small but interesting programs. Table 6 summarizes the programs we tested, with number of configurations created, and running times. Running times were measured using Sun's JVM1.2.2 for Windows NT, running on a 600MHZ Pentium III.

In our prototype, the conservative effect of an action is implemented in terms of the *focus* and *coerce* operations. Due to space limitations we can not elaborate, and the reader is referred to [33].

The *swap* and *swap_ord* programs use two threads swapping items in a linked list. *swap* does not use resource ordering, and thus may deadlock, *swap_ord* uses resource ordering, and thus can not deadlock. *stack* and *sStack* are non-synchronized and synchronized versions of a Stack ADT manipulated by multiple threads. *mutex* is a simple program using mutual exclusion to protect a critical section. *prodcons* and *sProdCons* are implementations of Queue ADT manipulated by producer and consumer threads. The *twoLockQ* is an implementation of the two-lock queue presented in [26] which is verified to preserve the invariants described in the manual proof of correctness in [26]. The *DP* program is an implementation of the *dining philosophers* problem with unbound number of philosopher threads.

7 Conclusion and Further Work

We have presented a parametric framework for verifying safety properties of concurrent Java programs. Our framework is a generalization of existing model-checking tech-

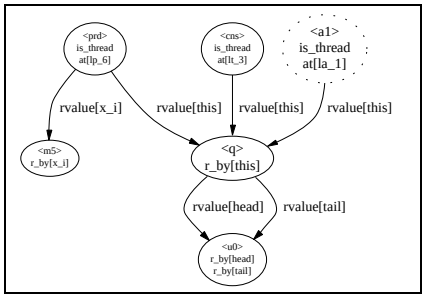

Figure 9: An abstract configuration C_9 in which interference between the consumer and the producer is detected.

Problem	Action	Precondition	Warning
Multiple starts	$v.start()$	$rvalue[v](t_r, dt) \wedge ts_running(dt)$	$IllegalThreadStateException$
		$rvalue[v](t_r, dt) \wedge ts_dead(dt)$	Dead thread can not be re-started
Premature stop	$v.stop()$	$rvalue[v](t_r, dt) \wedge ts_created(dt)$	Thread stopped before started
Missing ownership	$v.wait()$	$rvalue[v](t_r, l) \wedge \neg held_by(l, t)$	$IllegalMonitorStateException$
	$v.notify()$	$rvalue[v](t_r, l) \wedge \neg held_by(l, t)$	$IllegalMonitorStateException$
		$rvalue[v](t_r, l) \wedge \neg\exists(t_w) : waiting(t_w, l)$	A notify was ignored
Premature join	$v.join()$	$rvalue[v](t_r, dt) \wedge ts_created(dt)$	Thread join before started
Late setDaemon	$v.setDaemon()$	$rvalue[v](t_r, dt) \wedge ts_running(dt)$	$IllegalMonitorStateException$

Table 5: Preconditions for checking illegal and suspicious thread interactions.

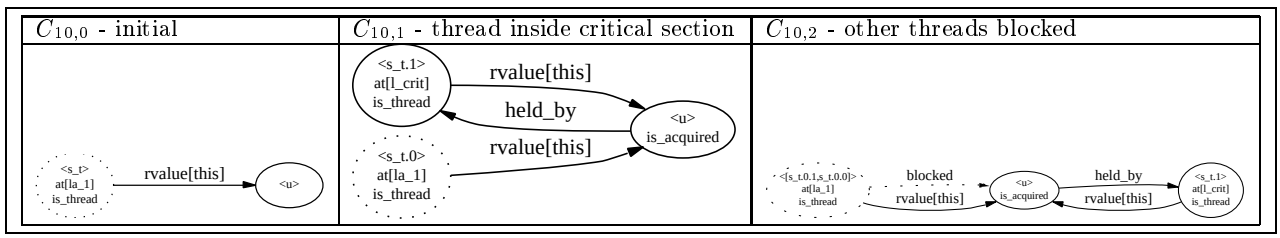

Figure 10: Configurations arising in mutual exclusion with an unbounded number of threads.

niques. The framework allows verification of multithreaded programs manipulating heap-allocated objects, and does not put a bound on the number of allocated objects.

We intend to exploit *partial order reduction* techniques such as [20] in order to improve scalability of our analysis.

Additional improvement may be gained by using symbolic representation of configurations using OBDDs [2, 3].

We believe that our framework can be extended to allow verification of general temporal specification other than safety properties.

References

[1] J. Aldrich, C. Chambers, E.G. Sirer, and S. Eggers. Static analyses for eliminating unnecessary synchronization from Java programs. In Agostino Cortesi and Gilberto Filé, editors, *Static Analysis*, volume 1694 of *Lecture Notes in Computer Science*, pages 19–38. Springer, 1999.

[2] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[3] Randal E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. Technical Report CS-92-160, Carnegie Mellon University, School of Computer Science, July 1992.

[4] L. Cardelli and A.D.Gordon. Mobile ambients. In *Proc. FoSSaCS'98, vol. 1378 of LNCS*, pages 140–155. Springer, 1998.

[5] D.R. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 296–310, New York, NY, 1990. ACM Press.

[6] E. M. Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks. *ACM Transactions on Programming Languages and Systems*, 19(5):726–750, September 1997.

[7] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[8] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

Program	Description	Config.	Time (sec)
swap	swapping list elements	16	10
swap_ord	swapping list elements with resource ordering	1	12
stack	interference on a non-synchronized stack	184	304
sStack	synchronized stack	104	330
mutex	mutual exclusion with unbound threads	33	2
nestedMon	nested monitors	42	7
prodCons	producer consumer	416	68
sProdCons	synchronized producer consumer	195	48
twoLockQ	two-lock queue	74	14
DP	dining philosophers unbound threads	514	23

Table 6: Number of configurations, and running times for the programs analyzed.

[9] J. Corbett. Using shape analysis to reduce finite-state models of concurrent java programs. October 1998.

[10] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, R. Shawn, and L. Hongjun. Bandera: Extracting finite-state models from Java source code. In *Proc. 22nd Int. Conf. on Soft. Eng. (ICSE)*, June 2000.

[11] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Symp. on Princ. of Prog. Lang.*, pages 238–252, New York, NY, 1977. ACM Press.

[12] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Symp. on Princ. of Prog. Lang.*, pages 269–282, New York, NY, 1979. ACM Press.

[13] C. Demartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent Java programs. *Software: Practice and Experience*, 29(7):577–603, June 1999.

[14] C. Demartini, R. Iosif, and R. Sisto. dSPIN : A dynamic extension of SPIN, September 1999.

[15] E. Emerson and A. P. Sistla. Symmetry and model checking. In C. Courcoubetis, editor, *Proc. of The Fifth Workshop on Computer-Aided Verificaton*, June/July 1993.

[16] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, 1997.

[17] K. Havelund and T. Pressburger. Model checking Java programs using Java pathfinder. *Int. Journal on Software Tools for Technology Transfer (STTT)*, 2(4), April 2000.

[18] C.A.R. Hoare. Recursive data structures. *Int. J. of Comp. and Inf. Sci.*, 4(2):105–132, 1975.

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

[20] G. J. Holzmann and D. Peled. An improvement in formal verification. In *Proc. Formal Description Techniques, FORTE94*, pages 197–211, Berne, Switzerland, October 1994. Chapman & Hall.

[21] A. Knapp, P. Cenciarelli, B. Reus, and M. Wirsing. An event-based structural operational semantics of multi-threaded java, 1998.

[22] D. Lea. *Concurrent Programming in Java*. Addison-Wesley, Reading, Massachusetts, 1997.

[23] T. Lev-Ami and M. Sagiv. TVLA: A framework for Kleene based static analysis. In *SAS'00, Static Analysis Symposium*. Springer, 2000. Available at http://www.math.tau.ac.il/~tla.

[24] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, January 1997.

[25] S.P. Masticola and B.G. Ryder. Non-concurrency analysis. *ACM SIGPLAN Notices*, 28(7):129–138, July 1993.

[26] M.M. Michael and M.L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC '96)*, pages 267–275, New York, USA, May 1996. ACM.

[27] G. Naumovich and G.S. Avrunin. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. In *Proceedings of the ACM SIGSOFT 6th International Symposium on the Foundations of Software Engineering (FSE-98)*, volume 23, 6 of *Software Engineering Notes*, pages 24–34, New York, November 3–5 1998. ACM Press.

[28] G. Naumovich, G.S. Avrunin, and L.A. Clarke. Data flow analysis for checking properties of concurrent Java programs. In *Proc. of the 1999 Int. Conf. on Soft. Eng.*, pages 399–410. IEEE Computer Society Press / ACM Press, 1999.

[29] R.H.B. Netzer and B.P. Miller. What are race conditions? some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, March 1992.

[30] F. Nielson, H. Riis Nielson, and M. Sagiv. A Kleene analysis of mobile ambients. In *Proceedings of ESOP'2000*, 2000.

[31] N. Rinetskey. Interprocedural shape analysis for recursive programs. Master's thesis, Technion, Israel, 2000. Available at http://www.cs.technion.ac.il/ maon/.

[32] R. Rugina and M. Rinard. Pointer analysis for multithreaded programs. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 77–90, Atlanta, Georgia, May 1–4, 1999.

[33] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Symp. on Princ. of Prog. Lang.*, 1999.

[34] H. Saidi. Model checking guided abstraction and analysis. In *Proceedings of the 7th International Static Analysis Symposium (SAS '00)*, 2000.

[35] A. Silberschatz and P. B. Galvin. *Operating Systems Concepts*. Addison-Wesley, Reading, 4 edition, 1994.

[36] S.D. Stoller. Model-checking multi-threaded distributed Java programs. In *Proc. of the 7th International SPIN Workshop on Model Checking of Software*, volume 1885 of *Lecture Notes in Computer Science*, pages 224–244. Springer-Verlag, August 2000.

[37] C. Ungureanu and S. Jagannathan. Concurrency analysis for java. In *Proceedings of the 7th International Static Analysis Symposium (SAS '00)*, 2000.

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

[39] E. Yahav. Verifying safety properties of concurrent java programs using 3-valued logic. Master's thesis, Tel-Aviv University, 2000. Available at http://www.math.tau.ac.il/∼yahave.

A Operational Semantics

This appendix gives an operational semantics for the Java statements not covered by the running example in the paper.

B Update Formulae for Instrumentation Predicates

C Analyzed Java Programs

```
public void swapContents(QueueItem other) {
  synchronized(this) {
    syncronized(other) {
      tempValue = other.value;
      other.value = value;
      value = tempValue;
    }
  }
}
```

Figure 11: Swapping the contents of queue items.

```
public void swapContents(QueueItem other) {
  if( this.hashCode() < other.hashCode()) {
    synchronized(this) {
      syncronized(other) {
        tempValue = other.value;
        other.value = value;
        value = tempValue;
      }
    }
  } else {
    synchronized(other) {
      syncronized(this) {
        tempValue = other.value;
        other.value = value;
        value = tempValue;
      }
    }
  }
}
```

Figure 12: Swapping the contents of queue items with resource ordering.

Action	Precondition	Predicate-update
$lock(v)$	$\neg\exists t' \neq t_r :$ $rvalue[v](t_r, l) \wedge held_by(l, t')$	$held_by(l_1, t_1) = held_by(l_1, t_1) \vee (t_1 = t_r \wedge l_1 = l)$ $blocked(t_1, l_1) = blocked(t_1, l_1) \wedge ((t_1 \neq t_r) \vee (l_1 \neq l))$
$unlock(v)$	$rvalue[v](t_r, l)$	$held_by(l_1, t_1) = held_by(l_1, t_1) \wedge (t_1 \neq t_r \vee l_1 \neq l)$
$wait(v)$	$rvalue[v](t_r, l)$	$held_by(l_1, t_1) = held_by(l_1, t_1) \wedge (t_1 \neq t_r \vee l_1 \neq l)$ $waiting(t_1, l_1) = waiting(t_1, l_1) \vee (t_1 = t_r \wedge l_1 = l)$
$notify(v)$	$rvalue[v](t_r, l) \wedge \exists t_w : waiting(t_w, l)$	$blocked(t_1, l_1) = blocked(t_1, l_1) \vee (t_1 = t_w \wedge l_1 = l)$ $waiting(t_1, l_1) = waiting(t_1, l_1) \wedge (t_1 \neq t_w \vee l_1 \neq l)$
$ignoredNotify(v)$	$rvalue[v](t_r, l) \wedge \neg\exists t_w : waiting(t_w, l)$	
$notifyAll(v)$	$rvalue[v](t_r, l) \wedge \exists t_w : waiting(t_w, l)$	$blocked(t_1, l_1) = blocked(t_1, l_1) \vee (waiting(t_1, l_1) \wedge (l_1 = l))$ $waiting(t_1, l_1) = waiting(t_1, l_1) \wedge (l_1 \neq l)$
$ignoredNotifyAll(v)$	$rvalue[v](t_r, l) \wedge \neg\exists t_w : waiting(t_w, l)$	
$isNull(v)$	$\neg\exists o_t : rvalue[v](t, o_t)$	
$isNotNull(v)$	$\exists o_t : rvalue[v](t_r, o_t)$	
$assignNull(v)$		$rvalue[v](t_1, o) =$ $rvalue[v](t_1, o) \wedge (t_1 \neq t_r)$ $r_by[fld](o_1) =$ $r_by[fld](o_1) \wedge$ $\exists(o_p, dst) : ((o_p \neq t_r) \wedge$ $rvalue[fld](o_p, dst) \wedge$ $rvalue[fld](t_r, dst))$
$assignReference(v_1, v_2)$	$rvalue[v_2](t_r, o_t)$	$rvalue[v_1](t_1, o_1) = rvalue[v_1](t_1, o_1) \vee ((t_1 = t_r) \wedge (o_1 = o_t))$ $r_by[v_1](o_1) = r_by[v_1](o_1) \vee (o_1 = o_t)$
$assignNew(v)$		$rvalue[v](t_1, o_t) = rvalue[v](t_1, o_t) \vee ((t_1 = t_r) \wedge isNew(o_t))$ $r_by[v](o_1) = r_by[v](o_1) \vee isNew(o_1)$
$selectField(v, o, f)$	$rvalue[o](t_r, o_t) \wedge rvalue[f](o_t, dst)$	$rvalue[v](t_1, o_1) = rvalue[v](t_1, o_1) \vee ((t_1 = t_r) \wedge (o_1 = dst))$
$updateField(o, f, v)$	$rvalue[o](t_r, o_t) \wedge rvalue[v](t_r, dst)$	$rvalue[f](o_s, o_d) = rvalue[f](o_s, o_d) \vee ((o_s = o_t) \wedge (o_d = dst))$ $r_by[f](o_1) = r_by[f](o_1) \vee (o_1 = dst)$
$blockLock(v)$	$\exists t' \neq t_r : rvalue[v](t_r, l) \wedge held_by(l, t')$	$blocked(t_1, l_1) = blocked(t_1, l_1) \vee (t_1 = t_r \wedge l_1 = l)$

Table 7: Operational semantics for concurrent Java with reference manipulation.

Action	Precondition	Predicate-update
$lock(v)$	$\neg\exists t' \neq t_r :$ $rvalue[v](t_r, l) \wedge held_by(l, t')$	$is_acquired(l_1) = is_acquired(l_1) \vee (l_1 = l)$ $is_blocked(t_1) = is_blocked(t_1) \wedge (t_1 \neq t)$ $slock(t_1) = slock(t_1) \wedge (t_1 \neq t)$ $wait_for(t_1, t_2) = wait_for(t_1, t_2) \vee ((t_2 = t) \wedge blocked(t_1, l))$
$unlock(v)$	$rvalue[v](t_r, l)$	$is_acquired(l_1) = is_acquired(l_1) \wedge (l_1 \neq l)$ $slock(t_1) = slock(t_1) \wedge (t_1 \neq t)$ $wait_for(t_1, t_2) = wait_for(t_1, t_2) \wedge ((t_2 \neq t) \wedge blocked(t_1, l))$
$wait(v)$	$rvalue[v](t_r, l)$	$is_acquired(l_1) = is_acquired(l_1) \wedge (l_1 \neq l)$ $is_waiting(t_1) = is_waiting(t_1) \vee (t_1 = t)$
$notify(v)$	$rvalue[v](t_r, l) \wedge \exists t_w : waiting(t_w, l)$	$is_blocked(t_1) = is_blocked(t_1) \vee (t_1 = t_w)$ $is_waiting(t_1) = is_waiting(t_1) \wedge (t_1 \neq t_w)$
$ignoredNotify(v)$	$rvalue[v](t_r, l) \wedge \neg\exists t_w : waiting(t_w, l)$	
$notifyAll(v)$	$rvalue[v](t_r, l) \wedge \exists t_w : waiting(t_w, l)$	$is_waiting(t_1) = is_waiting(t_1) \wedge \exists l_w : (l_w \neq l) \wedge waiting(t_1, l_w)$ $is_blocked(t_1) = is_blocked(t_1) \vee (waiting(t_1, l_1) \wedge (l_1 = l))$
$ignoredNotifyAll(v)$	$rvalue[v](t_r, l) \wedge \neg\exists t_w : waiting(t_w, l)$	
$blockLock(v)$	$\exists t' \neq t_r : rvalue[v](t_r, l) \wedge held_by(l, t')$	$is_blocked(t_1) = is_blocked(t_1) \vee (t_1 = t)$ $slock(t_1) = slock(t_1) \vee ((t_1 = t)$ $\wedge \bigcup_{v_1 \in Fields-v} (\exists l_v : rvalue[v_1](t_1, l_v) \wedge held_by(l_v, t_1) \wedge$ $fidlt[v, v_1](t)))$ $wait_for(t_1, t_2) = wait_for(t_1, t_2) \vee ((t_1 = t) \wedge held_by(t_2, l))$

Table 8: Predicate update formulae for instrumentation predicates.

```
public class Stack {
  private StackItem top;
  public void push(Object value) {
    StackItem x_i = new StackItem(value);
    synchronized(this) {
      if(top == null) {
        top = x_i;
      } else {
        x_i.next = top;
        top = x_i;
      }
    }
  }
  public Object pop() {
    Object value = null;
    StackItem x_d;
    synchronized(this) {
      if(top != null) {
        newTop = top.next;
        x_d = top;
        top = newTop;
        value = x_d.value;
      }
    }
    return value;
  }
}
```

Figure 13: Stack.

```
public void put(int value) {
     QueueItem x_i = new QueueItem(value);
     synchronize(this) {
         if (tail == null) {
            tail = x_i;
            head = x_i;
         } else {
            tail.next = x_i;
            tail = x_i;
         }
     }
}
public QueueItem take() {
     synchronized(this) {
         QueueItem x_d;
         if (head != null) {
            newHead = head.next;
            x_d = head;
            x_d.next = null;
            head = newHead;
            if (newHead == null) {
               tail = null;
            }
         }
     }
return x_d;
}
```

Figure 14: Queue implementation used by producer / consumer.

```
public class Philosopher extends Thread {
  private Fork leftFork,rightFork;
  public Philosopher(Fork left,Fork right) {
    leftFork = left;
    rightFork = right;
    }
  public void run() {
    synchronized(leftFork) {
      synchronized(rightFork) {
        eat();
      }
    }
  }
}
```

Figure 15: Dining Philosopher.

```
// TwoLockQueue.java
public class TwoLockQueue {
  private QueueItem head;
  private QueueItem tail;
  private Object headLock;
  private Object tailLock;
  ...
  public void put(Object value) {
       QueueItem x_i = new QueueItem(value);
       synchronize(tailLock) {
           tail.next = x_i;
           tail = x_i;
       }
  }
  public Object take() {
       synchronized(headLock) {
           Object x_d = null;
           QueueItem first=head.next;
           if (first != null) {
              x_d = first.value;
              head = first;
              head.value = null;
           }
       }
       return x_d;
  }
  }
```

Figure 16: Simplified Java source code for a two-lock-queue implementation [26].