Confirmation of Deadlock Potentials Detected by Runtime Analysis

Saddek Bensalem
Verimag
2, Avenue de Vignate
Gieres, France
bensalem@imag.fr

Jean-Claude Fernandez
Verimag
2, Avenue de Vignate
Gieres, France
fernand@imag.fr

Klaus Havelund
Kestrel Technology
4984 El Camino Real
Los Altos, California, USA
havelund@gmail.com

Laurent Mounier
Verimag
2, Avenue de Vignate
Gieres, France
mounier@imag.fr

ABSTRACT

This paper presents a framework for confirming deadlock potentials detected by runtime analysis of a single run of a multi-threaded program. The multi-threaded program under examination is instrumented to emit lock and unlock events. When the instrumented program is executed, a trace is generated consisting of the lock and unlock operations performed during that specific run. A lock graph is constructed which can reveal deadlock potentials in the form of cycles. The effectiveness of this analysis is caused by the fact that successful non-deadlocking runs yield as good, and normally better, information as deadlocking runs. Each cycle is then used to construct an observer that can detect the occurrence of the corresponding real deadlock, should it occur during subsequent test runs; and a controller, which, when composed with the program, determines the optimal scheduling strategy that will maximize the probability for the corresponding real deadlock to occur. The framework is formalized in terms of transition systems and is implemented in Java.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*monitors, testing tools, tracing*; D.3.2 [**Programming Languages**]: Language Classifications—*concurrent languages, Java*; D.4.1 [**Operating Systems**]: Process Management—*concurrency, deadlocks, scheduling, synchronization, threads*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*operational semantics, program analysis*

General Terms

Verification, Reliability, Algorithms, Theory.

Keywords

Deadlock detection, dynamic program analysis, false positives, Java, multi-threading, scheduler synthesis, testing.

1. INTRODUCTION

1.1 Background

Deadlocks form one of the important error categories of concurrent computer systems. This paper presents a technique for detecting deadlock potentials and for confirming the real deadlocks corresponding to these warnings. Normally one distinguishes between two kinds of deadlocks [19, 15]: *resource deadlocks* and *communication deadlocks*. A set of processes, or threads, is resource deadlocked if each process in the set requests a resource, a lock, held by another process in the set, forming a cycle of lock requests. In communication deadlocks, messages are the resources for which processes wait. In this paper we focus only on *resource deadlocks*, from now on referred to as deadlocks.

Locks are used to protect data against data races where several threads access shared objects simultaneously. Data races form the other problematic error category of concurrent systems and the avoidance of data races unfortunately frequently causes deadlocks when the locking discipline used results in cyclic lock acquisitions. The difficulty in detecting deadlocks comes from the fact that concurrent programs typically are non-deterministic: several executions of the same program on the same input may yield different behaviors due to slight differences in the way threads are scheduled.

The presented work builds on the previous work presented in [3], in which a very effective algorithm and a system were described for detecting deadlock potentials in multi-threaded Java programs. The algorithm described in [3] fundamentally examines a single execution trace, or program run, where *lock* and *unlock* events have been recorded, for example in a log file. On the basis of these locking events a lock graph is built, which with high probability will contain cycles in case deadlock potentials are present. That is, a cycle in the lock graph indicates that the program could potentially deadlock. However, this is not known for sure, which is also why the approach is unsound (a found error potential does not need to reflect a real error) and incomplete (errors may be missed). In this paper we augment the algorithm described in [3] with an extra phase that attempts to force the real deadlocks to occur based on the potentials identified. Although this still does not make the approach sound (nor complete), it does contribute to the information the programmer is given about the lock discipline of his/her program.

1.2 Approach

The algorithm presented works as follows. The runtime analysis algorithm described in [3] is first applied to a run of the program, where lock and unlock events are examined and recorded in a lock graph. Cycles in the lock graph represent deadlock potentials. Each such cycle, printed out in a readable format or visualized, might be enough information for the programmer to determine whether it represents a real problem or not. However, the result may be doubted, and we will then want to observe an actual deadlock corresponding to the suggested potential. The augmentation described in this paper does exactly that. The cycle forms the basis for constructing a test harness consisting of an *observer* that detects when the program actually deadlocks, and a *controller* that tries to steer the program into the deadlock suggested by the cycle. The program is then instrumented a second time, this time to interact with the observer and the controller: the controller is asked for permission to perform lock and unlock operations, and the observer is told which locks are locked and unlocked. The controller essentially attempts to prioritize lock operations performed by the threads involved in cycles in such a manner that the probability of a deadlock situation is increased. The framework is formalized in terms of transition systems and is implemented in Java. However, the methodology is language independent and can for example as well be applied to C and C++ programs that use POSIX threads [16].

1.3 Related work

1.3.1 *Detection of deadlock potentials*

A basic version of the runtime analysis algorithm described in [3] was previously implemented in the commercial tool Visual Threads [13]. This basic algorithm constructs a lock graph and detects cycles from a single non-deadlocking execution run. However, the basic algorithm can give false positives, putting a burden on the user to refute such. Our work in [3] augmented this work by minimizing the false positives in three cases: cycles within one thread, cycles under gate locks (a common lock taken first by all involved threads), and cycles between thread segments that cannot possibly execute in parallel. In yet earlier work we presented the GoodLock algorithm [10] which also attempts to improve the basic lock graph algorithm presented in [13] by reducing false positives in the presence of gate locks. This algorithm was based on building acyclic lock trees (rather than cyclic lock graphs as in [13]) but was limited to the detection of deadlocks between pairs of threads, rather than between any number of threads as in [3]. The latter and more recent solution builds directly on the cyclic graph model in [13]. In [21] is described an algorithm that extends the GoodLock algorithm for reducing false positives in presence of gate locks with the capability of detecting deadlock potentials between any number of treads (and not just two). This algorithm uses a combination of acyclic lock trees and cyclic lock graphs to represent locking patterns in a program run. In that work a framework is furthermore suggested for using static analysis in combination with dynamic analysis to detect deadlocks.

In these pieces of work a single execution trace is used as basis for the dynamic analysis. The ConTest tool [8] is able to analyze several execution traces, generated from several runs of the program being tested, creating a single locking model that is then analyzed. This approach is useful to reduce false negatives (missed errors).

Static analysis, such as performed by tools like [21], JLint [1], PolySpace [18] and ESC [6], analyze the source code without executing it. Static analysis can be very efficient and is generally preferable to dynamic analysis and testing techniques since one does not need to run the program, reducing the burden to compile and establish test cases. However, static analysis yields even more false positives and additionally cannot well analyze programs where the object structure is very dynamic. As described in [21] the ideal solution is a combination of static and dynamic analysis where each part supports the other.

1.3.2 *Confirmation of deadlock potentials*

In [10] an approach is described to detect deadlock potentials using runtime analysis and confirm them using model checking. The runtime analysis as well as confirmation algorithms were programmed as modifications to the Java PathFinder software model checker [22]. The runtime analysis would detect deadlock potentials and create a set of warnings. This information was then used to focus the model checker to only consider threads involved in the warnings, or having direct influence on such threads, and all possible interleavings would be tried for these involved threads. The approach presented in this paper attempts to approach this from a testing perspective, avoiding the need for a model checker.

Model checking has generally recently been applied directly to software, including the Java PathFinder system [11, 22] and other similar systems [9, 14, 4, 2, 20, 17, 7]. A model checker explores all possible execution paths of the program, and will therefore theoretically eventually expose a potential deadlock. This process is, however, quite resource demanding, in memory consumption as well in execution time, for large realistic programs.

The ConTest tool [8] attempts to expose deadlocks by inserting `sleep` and `yield` statements at specific locations in the code of the program under test. This is a technique to try and influence the behavior of the scheduler and hence be able to explore more execution scenarios, thereby increasing the chance of detecting a deadlock. The tool does, however, not use reports on deadlock potentials to steer the program under test into corners with high probability of deadlocking.

1.4 Outline of paper

Section 2 outlines the methodology of the approach and the involved components. Section 3 introduces preliminary concepts and notation used throughout the rest of the paper. Section 4 describes how from a cycle representing a deadlock potential to construct an observer that can detect the occurrence of the corresponding real deadlock, should it occur during test runs. Section 5 describes how to construct the controller which acts as a scheduler, how to instrument the program under test, and how to execute the instrumented program, the observer and the controller together. Section 6 discusses implementation issues and experimental results, while Section 7 concludes the paper.

2. METHODOLOGY

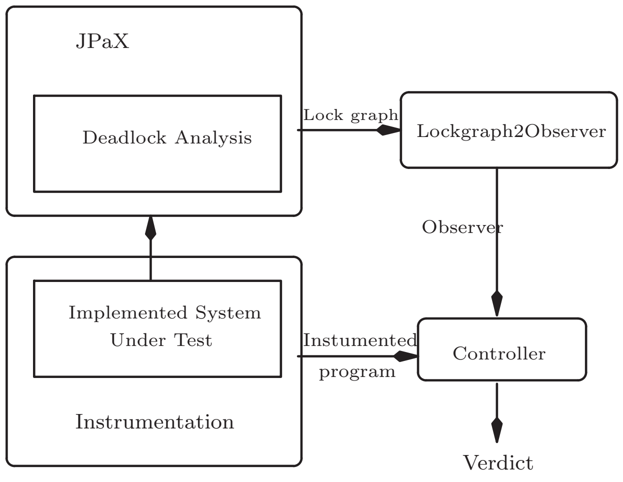


Figure 1: Methodology

Our methodology is illustrated in figure 1. It consists of the following phases:

1. Automatic detection of deadlock potentials in a multi-threaded program, the system under test (SUT), by examining a single execution trace.
2. Automatic generation of an observer for each cycle in the resulting lock graph.
3. Instrumentation of the system under test for the purpose of confirming deadlock potentials.
4. Search for "incorrect execution traces", by performing multiple "controlled runs".

We now elaborate on each of the phases in what follows. The first step is to detect deadlock potentials in the implemented system under test. The JPAX deadlock analyser [3] consists of two main modules, an instrumentation module and an observer module. The instrumentation module automatically instruments the bytecode class files of the multi-threaded program under test by adding new instructions that when executed generate the execution trace consisting of *lock* and *unlock* events needed for the analysis. The observer module reads the event stream and performs the deadlock analysis. That is, the instrumented program under observation is then executed, while the lock and unlock events are observed. A graph of locks is built, with edges between locks symbolizing locking orders. Any cycle in the graph signifies a deadlock potential. Having obtained the lock graph, the second step consists in generating automatically an observer for each cycle in the lock graph. The observer is a testing device. It observes the system under test. The last step is to control the execution of the SUT. If this execution is "accepted" by the observer, it means that it contains a deadlock (acceptance here means error detection).

3. PRELIMINAIRIES

3.1 General notations

A Labelled Transition System (LTS for short) is a quadruplet $M = (Q^{\mathrm{M}}, A^{\mathrm{M}}, R^{\mathrm{M}}, q^{\mathrm{M}}_{init})$ where Q^{M} is the finite set of states, A^{M} is a finite alphabet of actions, $R^{\mathrm{M}} \subseteq Q^{\mathrm{M}} \times A^{\mathrm{M}} \times Q^{\mathrm{M}}$ is the transition relation, and q^{M}_{init} is the initial state. We write $p \xrightarrow{a}_{R^{\mathrm{M}}} q$ iff $(p, a, q) \in R^{\mathrm{M}}$. A (finite) *execution sequence* σ of M is a sequence $q^{\mathrm{M}}_{init} \xrightarrow{a_1}_{R^{\mathrm{M}}} q_1 \ldots \xrightarrow{a_n}_{R^{\mathrm{M}}} q_n$. The sequence $w = a_1a_2 \ldots a_n$ of $A^{\mathrm{M}*}$ is called the *trace* of σ ($w = trace(\sigma)$), and state q_n is called the *ending state* of σ ($q_n = end(\sigma)$). The set of all execution sequences of M is denoted by $Exec(M)$.

An LTS M is said to be *deterministic* when for all states p and for all actions a, $p \xrightarrow{a}_{R^{\mathrm{M}}} q_1$ and $p \xrightarrow{a}_{R^{\mathrm{M}}} q_2$ implies $q_1 = q_2$. It is said to be *complete* when for all states p and for all actions a, there exists a state q such that $p \xrightarrow{a}_{R^{\mathrm{M}}} q$.

For a set E, $\vec{x}_n$ (or simply $\vec{x}$ when value n is clear from the context) denotes a tuple $(x_1, x_2, \ldots x_n)$ of E^n. For i in $[1, n]$, $\vec{x}(i)$ represents the value x_i (the i^{th} element of $\vec{x}$), and $\vec{x}[i \leftarrow x'_i]$ denotes the tuple $(x_1, x_2, \ldots x'_i, \ldots x_n)$, obtained from $\vec{x}$ by replacing its i^{th} element by x'_i. Finally, for any element e of E, $\vec{e}$ represents the tuple $(e, e, \ldots, e)$.

3.2 Program models and observers

A Java program that we consider can be viewed as a pair $(\mathcal{T}, \mathcal{O})$, where $\mathcal{T} = \{T_i \mid i \in \{1, 2, \cdots, n\}\}$ is a set of parallel threads T_i, that synchronize on a set of shared resources $\mathcal{O} = \{O_j \mid j \in \{1, 2, \cdots, p\}\}$. The behaviour of each thread T_i can be expressed by an LTS $S_{T_i} = (Q^{\mathrm{S}_{\mathrm{T}_i}}, A^{\mathrm{S}_{\mathrm{T}_i}}, R^{\mathrm{S}_{\mathrm{T}_i}}, q^{\mathrm{S}_{\mathrm{T}_i}}_{init})$ such that

$$\begin{aligned} A^{\mathrm{S}_{\mathrm{T}_i}} = \; & \{L(T_i, O_j) \mid i \in \{1, \cdots, n\} \wedge j \in \{1, \cdots, p\}\} \cup \\ & \{U(T_i, O_j) \mid i \in \{1, \cdots, n\} \wedge j \in \{1, \cdots, p\}\} \cup \\ & \{\tau\} \end{aligned}$$

Intuitively, action $L(T_i, O_j)$ (*resp.* $U(T_i, O_j)$) denotes a lock (*resp.* unlock) statement performed by T_i on resource O_j, and τ denotes any other action.

According to the semantics of *lock* and *unlock* actions, the behaviour of a program $\mathcal{P} = (\mathcal{T}, \mathcal{O})$ can be expressed by an LTS $S_{\mathcal{P}} = (Q, A, R, q_{init})$ such that: $Q \subseteq ((Q^{\mathrm{T}_1} \times \cdots \times Q^{\mathrm{T}_n}) \times (\mathcal{T} \cup \{\perp\})^p)$ (the second component is a vector that for each object O indicates the thread that holds this lock, or $\perp$ if no thread holds it), $A \subseteq (A^{\mathrm{T}_1} \cup \cdots \cup A^{\mathrm{T}_n})$, $q_{init} = (q^{\mathrm{T}_1}_{init}, \ldots, q^{\mathrm{T}_n}_{init}, \vec{\perp})$, and R is the smallest set verifying the following rules:

$$\frac{(\vec{p},\vec{o}) \in Q,\ \vec{p}(i) \xrightarrow{\tau}_{T^{T_i}} q_i}{(\vec{p},\vec{o}) \xrightarrow{\tau}_{R} (\vec{p}[i \leftarrow q_i],\vec{o})} \quad \text{[Rule 1]}$$

$$\frac{(\vec{p},\vec{o}) \in Q,\ \vec{p}(i) \overset{L(T_i,O_j)}{\longrightarrow}{}_{R^{T_i}}\ q_i,\ \vec{o}(j) = \bot}{(\vec{p},\vec{o}) \overset{L(T_i,O_j)}{\longrightarrow}{}_{R}\ (\vec{p}[i \leftarrow q_i],\vec{o}[j \leftarrow T_i])} \quad \text{[Rule 2]}$$

$$\frac{(\vec{p},\vec{o}) \in Q,\ \vec{p}(i) \overset{U(T_i,O_j)}{\longrightarrow}{}_{R^{T_i}}\ q_i,\ \vec{o}(j) = T_i}{(\vec{p},\vec{o}) \overset{U(T_i,O_j)}{\longrightarrow}{}_{R}\ (\vec{p}[i \leftarrow q_i],\vec{o}[j \leftarrow \bot])} \quad \text{[Rule 3]}$$

A (partial) *deadlock* of $\mathcal{P}$ is a state p of $S_{\mathcal{P}}$ in which a subset of threads $\mathcal{T}$ are mutually waiting for a same set of resources to be freed. It is formalized in the following definition:

Definition 1 (Deadlock states). *Let $\mathcal{P}$ be a program and $S_{\mathcal{P}}$ its associated LTS. $\mathcal{P}$ contains a deadlock state, denoted by deadlock($\mathcal{P}$), iff there exists a state $p = (\vec{p},\vec{o})$ of $S_{\mathcal{P}}$ such that:*

$$\begin{array}{l}
\exists X \subseteq \{1,2,\dots n\} \cdot \forall i \in X \cdot \exists j \in \{1,2,\dots p\} \cdot \\
\quad \vec{p}(i) \overset{L(T_i,O_j)}{\longrightarrow}{}_{R^{T_i}}\ q_i \wedge \\
\quad \exists k \in X \cdot \vec{o}(j) = k \wedge \\
\quad \forall a \neq L(T_i,O_j) \cdot \\
\qquad \neg\exists q_i' \cdot \vec{p}(i) \xrightarrow{a}_{R^{T_k}} q_i'
\end{array}$$

Finally, an *observer* is an LTS equiped with a set of distinguished states (its *accepting* states). Execution sequences of the observer that end in an accepting state are said to be *accepted* by the observer:

Definition 2 (Observers). *An observer $\mathcal{OBS}$ is a pair (Obs, Accept) where Obs is a* deterministic *and* complete *LTS: $Obs = (Q^{Obs}, A^{Obs}, R^{Obs}, q^{Obs}_{init})$, and Accept is a set of distinguished states of Obs: $Accept \subseteq Q^{Obs}$.*

An observer $\mathcal{OBS} = (Obs, Accept)$ accepts *an execution sequence σ of a program $\mathcal{P}$ if and only if there exists an execution sequence σ_o of Obs with the same trace as σ and whose ending state belongs to Accept :*

$$\begin{array}{l}
accept(Obs,\sigma) \equiv_{def} \\
\quad \exists \sigma_o \in Exec(Obs) \cdot \\
\qquad trace(\sigma) = trace(\sigma_o) \wedge end(\sigma) \in Accept
\end{array}$$

4. FROM DEADLOCK POTENTIALS TO OBSERVERS

4.1 Deadlock detection

In Java, a thread can lock an object using the `synchronized` statement, or by declaring methods on the shared object `synchronized`, which is equivalent. For example, a thread `t` can obtain a lock on an object `A` and then execute a statement `S` while having that lock by executing the following statement: `synchronized(A){S}`. During the execution of `S`, no other thread can obtain a lock on `A`. The lock is unlocked when the scope of the synchronized statement is left; that is, when execution passes the curly bracket: '}'.

The algorithm for detecting deadlock potentials we use, extends an existing algorithm by reducing the amount of false positives reported [3], and has been implemented in the Java PathExplorer tool [12]. This algorithm is neither sound nor complete, but it scales and it is very effective: it finds bugs with high probability and it yields few false positives. In essence, the algorithm for detecting deadlock potentials works as follows. The multi-threaded program under observation is executed, while just lock and unlock events are observed. A graph of locks is built, with edges between locks symbolizing locking orders. An edge, in the lock graph, between two locks o_1 and o_2 and labelled by t, means that t owns o_1 while tacking o_2. Any cycle in the graph signifies a potential for a deadlock.

$Main$:

```
01: new T1().start();
02: new T2().start();
```

T_1 :

```
03: synchronized(G){
04:   synchronized(L1){
05:     synchronized(L2){}
06:   }
07: };
08: t3 = new T3();
09: t3.start();
10: t3.join();
11: synchronized(L2){
12:  synchronized(L1){}
13: }
```

T_2 :

```
14: synchronized(G){
15:   synchronized(L2){
16:     synchronized(L1){}
17:   }
18: }
```

T_3 :

```
19: synchronized(L1){
20:   synchronized(L2){}
21: }
```

Figure 2: Example containing four cycles, only one of which represents a deadlock potential

The main task performed by the detection algorithm is to find cycles among transactions, illustrating the potential for threads waiting for resources held by other threads in a cyclic manner. The detection of cyclic locking acquisitions can yield false positives that do not represent real deadlocks. Three categories of false positives can be avoided by the deadlock analyser in Jpax. The first category, *single threaded cycles*, refer to cycles that are created by one single thread. *Guarded cycles* refer to cycles that are guarded by a gate lock "taken higher" up by all involved threads. Finally, *thread segmented cycles* refer to cycles between thread segments that cannot possibly execute concurrently. For the example in Figure 2, the deadlock analyser Jpax generates only one real deadlock potential between threads T_2 and T_3, corresponding to a cycle on L_1 and L_2 (see lock graph in Figure 3). The three other cycles in this graph correspond to false positives, and the detection algorithm eliminates them. The single threaded cycle within T_1 clearly does not represent a deadlock. The guarded cycle between T_1 and T_2 does not represent a deadlock since both threads must acquire the gate lock G first. Finally, the thread segmented cycle between T_1 and T_3 does not represent a deadlock since T_3 will execute before T_1 executes its last synchronization segment.

4.2 From lock graph cycles to observers

The idea here is to build an observer from a cycle in the lock graph. The obtained observer will characterize all possible interleavings that lead to a deadlock state. This can

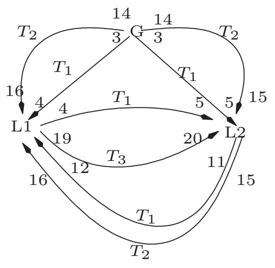


Figure 3: The lock graph of the example in Figure 2 (line numbers are added on the edges to make the correspondance with the statements in the program clear)

be done as follows. First, for each edge in a cycle of the lock graph we associate an (atomic) observer with two transitions. Then the parallel composition of all these atomic observers can be seen as the global observer that characterizes the potential deadlock.

Definition 3 (Atomic observer). *Let C be a cycle in the lock graph and $e_k = (o_i, T, o_j)$ the k^{th} edge in C. We associate an atomic observer $\mathcal{O}_k = (Obs_k, Accept_k)$ to the edge e_k, such that :*

- *$Obs_k = (Q^k, A^k, R^k, q^k_{init})$, where :*
  1. *Q^k, the set of states, is $\{2k-1, 2k\}$,*
  2. *A^k, the finite alphabet of actions, has two elements $L(T, o_j)$ and $U(T, o_j)$*
  3. *R^k has two transitions: $2k-1 \xrightarrow{L(T,o_j)} 2k$ and $2k \xrightarrow{U(T,o_j)} 2k-1$, and*
  4. *q^k_{init}, the initial state, is $2k-1$*
- *$Accept_k$ has one distinguished state $2k$*

The product of the atomic observers associated to the all edges of one cycle should represent all possible interleavings of the lock/unlock events from different threads in this cycle, respecting that locks can only be held by one thread at a time. The composed observer is defined as follows.

Definition 4 (Observer associated to a Cycle). *Let $\mathcal{C} = e_1, \ldots, e_n$ be a cycle in the lock graph and $\mathcal{O}_k = (Obs_k, Accept_k)$ for $k = 1, \ldots, n$ be the atomic obervers associated with each edge e_k in $\mathcal{C}$, where Obs_k is the quadruplet $(Q^k, A^k, R^k, q^k_{init})$. We define the observer $\mathcal{O}_\mathcal{C}$ associated to the cycle $\mathcal{C}$ by $(Obs_\mathcal{C}, Accept_\mathcal{C})$, such that :*

- *$Obs_\mathcal{C} = (Q^\mathcal{C}, A^\mathcal{C}, R^\mathcal{C}, q^\mathcal{C}_{init})$ is the composition of the labelled transition systems Obs_k for $k = 1, \ldots n$, denoted by $\|_{k=1}^n (Q^k, A^k, R^k, q^k_{init})$, where :*
  1. *$Q^\mathcal{C} = Q^1 \times Q^2 \times \ldots \times Q^n$,*
  2. *$A^\mathcal{C} = \bigcup_{k=1}^n A^k$,*
  3. *$R^\mathcal{C} \subseteq Q^\mathcal{C} \times A^\mathcal{C} \times Q^\mathcal{C}$ is defined by :*

$$\frac{q_i \xrightarrow{a} q'_i}{(q_1, \ldots, q_i, \ldots, q_n) \xrightarrow{a} (q_1, \ldots, q'_i, \ldots, q_n)}$$

  where a is an action denoting a lock $L(T_i, o_j)$ or an unlock $U(T_i, o_j)$ statement.

  4. *$q_{init}{}^\mathcal{C} = (q^1{}_{init}, q^2{}_{init}, \ldots, q^n{}_{init})$*
- *The set $Accept_\mathcal{C}$ of distinguished states is $Accept_1 \times Accept_2 \times \ldots \times Accept_n$*

Example 4.1. *Let us consider the real deadlock potential between threads T_2 and T_3 in the example in Figure 2, corresponding to the cycle $(L2, T_2, L1), (L1, T_3, L2)$. The atomic observers associated with each edge in the cycle are respectively :*

$$((\{1,2\}, \{L(T_2, L1), U(T_2, L1)\},$$
$$\{(1, L(T_2, L1), 2), (2, U(T_2, L1), 1)\}, 1), \{2\})$$

and

$$((\{3,4\}, \{L(T_3, L2), U(T_3, L2)\},$$
$$\{(3, L(T_3, L2), 4), (4, U(T_3, L2), 3)\}, 3), \{4\})$$

For the cycle the observer is $O_\mathcal{C} = (Obs_\mathcal{C}, \{2, 4\})$, where $Obs_\mathcal{C}$ is the transition system in Figure 4.

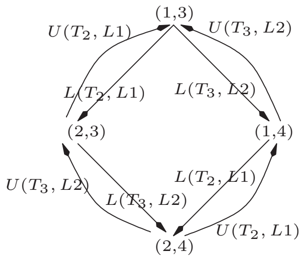


Figure 4: The transition system of observer $O_\mathcal{C}$

5. CONFIRMATION OF DEADLOCK POTENTIALS

In the previous section we saw how each cycle of the *lock graph* produced by Jpax can be turned into an *observer*, able to reveal the corresponding *real* deadlock in the program under verification, in case the potential is not a false positive and has such a real manifestation. In particular, if an execution sequence of this program is *accepted* by this observer, then this sequence contains a deadlock (i.e., a state in which a group of threads are waiting for mutual unlock actions on the same resource set).

5.1 Program execution guided by an observer

In this section we show how the program execution can be guided by this observer to increase the chance of exhibiting such a deadlock state. Instead of just monitoring the program with the observer, we propose to control its execution in order to favour the execution sequences that make progress towards an accepting state of the observer. This is achieved in the following way:

Instrumentation phase

First, the program under test is modified:

- A new controller thread is added, in charge of driving the program execution towards an observer's accepting state.
- Each program statement a (performed by a thread T) which is monitored by the observer (a is a lock or unlock instruction) is replaced by a two way

communication with the controller: $Req(a)$, request from the thread T to perform action a, followed by $Grant(a)$, response from the controller to grant this action. The thread T is blocked until reception of this grant, and it executes action a when it receives it (in an atomic way).

Execution phase

Then, this instrumented program is executed:

- The controller blocks all the requests it receives.
- When all monitored threads are blocked (or when a timer expires, to avoid an infinite wait), it liberates one of them to allow progress towards an accepting state. *unlock* requests are always granted first corresponding to a "backtrack" in the observer's state space. Otherwise *lock* requests are granted in such a manner as to give all threads involved in a deadlock cycle their *first* lock before any thread is granted permission to its *second* lock (which it then cannot get).

This approach is formalised in the next paragraph.

5.2 Formalization

The **instrumentation phase** can be modelled as an LTS transformation parameterized by an action set A, as explained in the definition below.

Definition 5 (Instrumented LTS). *Let $M = (Q^M, A^M, R^M, q^M_{init})$ be an LTS. The instrumented LTS M' of M with respect to the action set A, denoted by $instrument(M, A)$, is the LTS $(Q^{M'}, A^{M'}, R^{M'}, q^{M'}_{init})$ defined as follows. Let $R_A = \{(p, a, q) \mid a \in A \wedge p \xrightarrow{a}_{R^M} q\}$, $k = |R_A|$, and $q^A_1, \cdots q^A_k$ be fresh names of states not in Q^M. Then:*

- $Q^{M'} = Q^M \cup \{q^A_1, \cdots q^A_k\}$
- $A^{M'} = (A^M \setminus A) \cup \{\texttt{Req}(a) \mid a \in A\} \cup \{\texttt{Grant}(a) \mid a \in A\}$
- *We define $p \xrightarrow{\alpha}_{R^{M'}} q$:*
  - $p \xrightarrow{\alpha}_{R^{M'}} q$ *iff* $p \xrightarrow{\alpha}_{R^M} q$ *and* $\alpha \notin A$
  - *Let $R_A = \{t_1, \cdots, t_n\}$. For each $t_i : p \xrightarrow{a}_{R^M} q$ of R_A, using a fresh name ${q^A}_i$, we have :* $p \xrightarrow{\texttt{Req}(a)}_{R^{M'}} {q_i}^A \xrightarrow{\texttt{Grant}(a)}_{R^{M'}} q$, *where $(q^A_i, \texttt{Grant}(a), q)$ is the unique outgoing transition of q^A_i.*
- $q^{M'}_{init} = q^M_{init}$

Let $\mathcal{P} = (\mathcal{T}, \mathcal{O})$ be a program, and $Obs = (Obs, Accept)$ an observer. In the following we denote by $\mathcal{P}'$ the instrumented program obtained by replacing each LTS S_{T_i} by its instrumented version $S'_{T_i} = (Q^{\mathrm{S'T}_i}, A^{\mathrm{S'T}_i}, R^{\mathrm{S'T}_i}, {q_{init}}^{\mathrm{S'T}_i})$ with respect to the observer's action set A^{Obs}. That is, we have that $S'_{T_i} = instrument(S_{T_i}, A^{\mathrm{Obs}})$.

The **execution phase** consists in a program execution guided by a `Controller` thread towards an accepting state of the observer. Therefore the `Controller` thread should maintain the current observer's state (according to the actions executed so far by the program) and the status of the threads monitored by this observer (i.e., are they currently blocked on a lock or a unlock action). This status can be modeled by two tuples:

- $\vec{x} = (x_1, x_2, \ldots, x_n)$, where $x_i \in A^{\mathrm{Obs}}$ is the action requested by thread T_i, or is equal to ε if T_i is not currently blocked;
- $\vec{b} = (b_1, b_2, \ldots, b_n)$, where b_i is a boolean value equal to true if thread T_i is blocked and to false otherwise.

The `Controller` thread can be expressed by an LTS $S^{\mathrm{CTR}} = (Q^{\mathrm{CTR}}, A^{\mathrm{CTR}}, R^{\mathrm{CTR}}, q^{\mathrm{CTR}}_{init})$ such that:

- $Q^{\mathrm{CTR}} \subseteq Q^{\mathrm{Obs}} \times (A^{\mathrm{Obs}} \cup \{\varepsilon\})^n \times \mathbb{B}^n$
- $A^{\mathrm{CTR}} = A^{\mathrm{Obs}}$
- R^{CTR} is the smallest set obtained by applying the rules in Figure 5
- $q^{\mathrm{CTR}}_{init} = (q^{\mathrm{Obs}}_{init}, \vec{\varepsilon}, \vec{false})$

The resulting instrumented program behaviour during the **execution phase** can then be modelled by an LTS $S^{\mathrm{CTR}}_{\mathcal{P}} = (Q, A, R, q_{init})$ such that:

- $Q \subseteq ((Q^{\mathrm{S'T}_1} \times Q^{\mathrm{S'T}_2} \times \cdots \times Q^{\mathrm{S'T}_n}) \times (\mathcal{T} \cup \{\perp\})^p \times Q^{\mathrm{CTR}}) \cup \{\delta\}$
- $A \subseteq (A^{\mathrm{S'T}_1} \cup A^{\mathrm{S'T}_2} \cup \cdots \cup A^{\mathrm{S'T}_n})$
- R is the smallest set obtained by applying the rules in Figure 5,
- $q_{init} = ({q_{init}}^{\mathrm{S'T}_1}, \ldots, {q_{init}}^{\mathrm{S'T}_n}, \vec{\perp}, {q_{init}}^{\mathrm{CTR}})$

Intuitively the rules in Figure 5 can be interpreted as follows (an application example is presented in the next section):

- Rule R4 states that if an accepting state of the controller has been reached, then the current execution sequence terminates in the special deadlock state δ.
- Rules R5 and R6 correspond to a *request* received from thread T_i: this thread becomes blocked ($\vec{b}(i)$ is set to true), and the requested action is stored ($\vec{x}(i)$ is updated).
- Rules R7 and R8 correspond to a *grant* issued by the `Controller`: when all the threads are currently blocked ($\vec{b}$ is equal to $\vec{true}$), then if an *unlock* action can be performed, it is granted (rule R8), otherwise a *lock* action is granted (rule R7).

The rules are applied such that lock releases have highest priority and lock requests that do not maximize the probability of a deadlock have lowest priority. Those are the secondary lock requests that would block in a deadlock situation. The result will be that all threads take their first lock first, and now cannot get their second lock.

Finally, it is easy to prove that whenever a deadlock state is found during a guided execution, then it corresponds to a real deadlock in the original program (since the `Controller` only restricts the program behaviour):

Theorem 1. *If the LTS $S^{CTR}_{\mathcal{P}}$ contains an execution sequence that ends in the state δ, then program $\mathcal{P}$ contains a deadlock state :*

$$(\exists \sigma \in Exec(S^{CTR}_{\mathcal{P}}) \cdot end(\sigma) = \delta) \Rightarrow deadlock(\mathcal{P})$$

$$\frac{p^{\mathrm{Obs}} \in \mathit{Accept}}{(\vec{p}, \vec{o}, (p^{\mathrm{Obs}}, \vec{x}, \vec{b})) \xrightarrow{\tau}_{T} \delta} \quad \text{[Rule 4]}$$

$$\frac{\vec{p}(i) \xrightarrow{\texttt{Req}(L(T_i,O_j))}_{R^{\mathrm{S'}\mathrm{T}_i}} q_i}{(\vec{p}, \vec{o}, (p^{\mathrm{Obs}}, \vec{x}, \vec{b})) \xrightarrow{\texttt{Req}(L(T_i,O_j))}_{R} (\vec{p}[i \leftarrow q_i], \vec{o}, (p^{\mathrm{Obs}}, \vec{x}[i \leftarrow L(T_i,O_j)], \vec{b}[i \leftarrow \mathit{true}]))} \quad \text{[Rule 5]}$$

$$\frac{\vec{p}(i) \xrightarrow{\texttt{Req}(U(T_i,O_j))}_{R^{\mathrm{S'}\mathrm{T}_i}} q_i}{(\vec{p}, \vec{o}, (p^{\mathrm{Obs}}, \vec{x}, \vec{b})) \xrightarrow{\texttt{Req}(U(T_i,O_j))}_{R} (\vec{p}[i \leftarrow q_i], \vec{o}, (p^{\mathrm{Obs}}, \vec{x}[i \leftarrow U(T_i,O_j)], \vec{b}[i \leftarrow \mathit{true}]))} \quad \text{[Rule 6]}$$

$$\frac{\exists i.\ \vec{x}(i) = L(T_i,O_j),\ \vec{o}(j) = \bot,\ p^{\mathrm{Obs}} \xrightarrow{L(T_i,O_j)}_{T^{\mathrm{Obs}}} q^{\mathrm{Obs}}}{(\vec{p}, \vec{o}, (p^{\mathrm{Obs}}, \vec{x}, \vec{true})) \xrightarrow{\texttt{Grant}(L(T_i,O_j))}_{R} (\vec{p}[i \leftarrow q_i], \vec{o}[j \leftarrow T_i], (q^{\mathrm{Obs}}, \vec{x}[i \leftarrow \varepsilon], \vec{b}[i \leftarrow \mathit{false}]))} \quad \text{[Rule 7]}$$

$$\frac{\exists i.\ \vec{x}(i) = U(T_i,O_j),\ \vec{o}(j) = T_i,\ p^{\mathrm{Obs}} \xrightarrow{U(T_i,O_j)}_{T^{\mathrm{Obs}}} q^{\mathrm{Obs}}}{(\vec{p}, \vec{o}, (p^{\mathrm{Obs}}, \vec{x}, \vec{true})) \xrightarrow{\texttt{Grant}(U(T_i,O_j))}_{R} (\vec{p}[i \leftarrow q_i], \vec{o}[j \leftarrow \bot], (q^{\mathrm{Obs}}, \vec{x}[i \leftarrow \varepsilon], \vec{b}[i \leftarrow \mathit{false}]))} \quad \text{[Rule 8]}$$

Figure 5: Rules

6. IMPLEMENTATION ISSUES

The technique proposed in this paper to confirm the existence of potential deadlocks revealed by JPAX using guided execution is currently under implementation. However, a first prototype does exist, and its use has confirmed the utility of this approach. We discuss in this section some of the implementation issues, and give some experimental results obtained with the tool.

6.1 Implementation issues

According to the methodology described in the previous section, guiding a program towards the accepting state of an observer requires two successive phases: instrumentation of the original program (the program under test), and execution of this instrumented program in coordination with a `Controller` thread.

Instrumentation

The program instrumentation is performed at the *byte code* level using the BCEL byte code analysis and transformation tool [5]. The instrumentation phase consists of a traversal of the byte code program's abstract syntax tree while inserting the communications with the `Controller` before each relevant instruction, according to definition 5. Relevant instructions considered so far are entry and exit points of synchronisation blocks (`monitorenter` and `monitorexit` at the bytecode level).

Communications between the instrumented threads and the `Controller` are implemented by means of message exchanges using *sockets*. This solution allows to run the `Controller` thread on a remote machine, if necessary (i.e., if the test architecture does not allow to run it directly on the same platform as the program under test). Messages sent by a thread T_i to the `Controller` contains the following information: the thread id of T_i, the request type (*lock* or *unlock* actions corresponding to entry and exit of a synchronized block), and the identity of the object to lock or unlock, that could be either a method variable, a class variable or the object of a method (when the synchronized block corresponds to a synchronized method).

Execution with the `Controller` thread:

The `Controller` thread first reads as input an observer description and then starts the program under test and interacts with it. It maintains a set of data structures allowing to implement the rules R4 to R8 described in figure 5: the current observer state, the current set of locked threads ($\vec{b}$), and the current set of requested actions ($\vec{x}$). The `Controller` implementation is then rather straightforward: it receives request messages from the program under test, and behaves according to rules R4 to R8. However, to avoid an infinite wait of the `Controller` (when the running thread T_i will no longer perform any request action), a timer is used to indicate to the `Controller` to "liberate" another blocked thread (if any), applying rule R7 or R8, increasing the chance to reach an accepting state. Finally, the `Controller` can also store the current execution sequence to issue a kind of "diagnostic trace" if an accepting state of the observer is reached.

6.2 Some experimental results

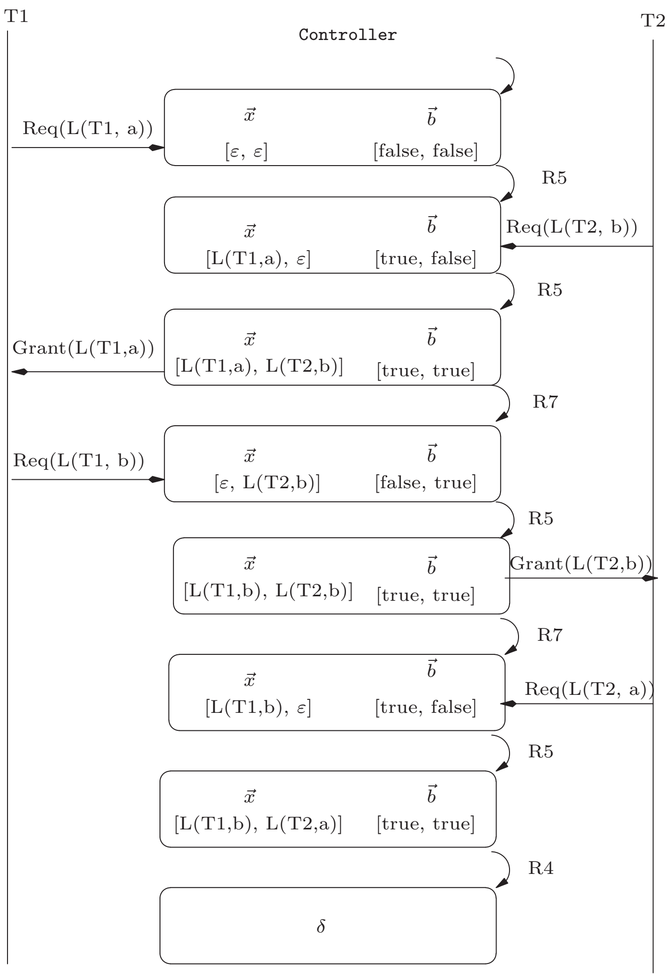
In our experiments with this first prototype we only considered rather simple Java programs. Our purpose here was only to identify some potential problems in our implementation, and to roughly evaluate its performance (in terms of execution time, and ability to detect deadlocks). Checking its scalability to larger examples would clearly require more experiments.

The pseudo-Java code below describes a program with two parallel threads `T1` and `T2` entering two nested synchronization blocks (associated to variables `a` and `b`) depending on the value of a static variable `x`.

```
class T implements Runnable {
  static int x=0;
   ...
  public void run() {
     while (true) {
       if (x == 0) {
          x=1;
          synchronized (a) {
              synchronized (b) {
                 ...
              }
          }
       } else {
          x=0;
          synchronized (b) {
              synchronized (a) {
                 ...
              }
          }
       }
     }
  }
}

public static void main(String[] args){
    Thread T1 = new Thread(new T());
    Thread T2 = new Thread(new T());
    T1.start();
    T2.start();
         ...
}
```

Executing this program may lead to a deadlock. Surpisingly this deadlock happens to be difficult to observe in practice, even when running this program on several platforms, or with several Java runtime environments. However it is immediately found when this program is instrumented and executed in conjunction with the `Controller` thread. A possible guided execution is illustrated in Figure 6.



T1
Controller
T2
Req(L(T1, a))
$\vec{x}$ $[\varepsilon, \varepsilon]$ $\vec{b}$ [false, false]
R5
Req(L(T2, b))
$\vec{x}$ [L(T1,a), ε] $\vec{b}$ [true, false]
R5
Grant(L(T1,a))
$\vec{x}$ [L(T1,a), L(T2,b)] $\vec{b}$ [true, true]
R7
Req(L(T1, b))
$\vec{x}$ [ε, L(T2,b)] $\vec{b}$ [false, true]
R5
Grant(L(T2,b))
$\vec{x}$ [L(T1,b), L(T2,b)] $\vec{b}$ [true, true]
R7
Req(L(T2, a))
$\vec{x}$ [L(T1,b), ε] $\vec{b}$ [true, false]
R5
$\vec{x}$ [L(T1,b), L(T2,a)] $\vec{b}$ [true, true]
R4
δ

Figure 6: Example of guided execution

7. CONCLUSION

A framework has been presented for detecting deadlocks in multi-threaded programs. The detection consists of two phases: a deadlock potential *identification* phase, and a deadlock potential *confirmation* phase. During the first phase the program is instrumented to emit locking events to a runtime analyzer, which builds a lock graph where cycles represent deadlock potentials. In the second phase the original program is again instrumented, this time to communicate with an observer that can detect the occurrence of real deadlocks and a controller that can drive the application into the deadlocks suggested by the cycles. The observer and the controller are synthesized from the cycles produced during the first phase. The framework has been formalized in terms of transition systems and has been implemented in Java. A small set of test cases have been performed, illustrating that deadlocks are found efficiently and effectively due to the first phase, and that true positives (real deadlocks) can be confirmed and demonstrated automatically. The approach is attractive since it does not require user provided input (beyond the program) and since the technique seems to scale very well. A combination with static analysis as described in [21] together with a random scheduling framework as described in [8] would be a very strong defense against deadlocks in realistically sized programs and will be pursued in future work.

8. REFERENCES

[1] C. Artho and A. Biere. Applying Static Analysis to Large-Scale, Multi-threaded Java Programs. In D. Grant, editor, *13th Australien Software Engineering Conference*, pages 68–75. IEEE Computer Society, August 2001.

[2] T. Ball, A. Podelski, and S. Rajamani. Boolean and Cartesian Abstractions for Model Checking C Programs. In *Proceedings of TACAS'01: Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, Genova, Italy, April 2001.

[3] S. Bensalem and K. Havelund. Dynamic Deadlock Analysis of Multi-threaded Programs. In Shmuel Ur, Eyal Bin, and Yaron Wolfsthal, editors, *Haifa Verification Conference*, volume 3875 of *LNCS*, pages 208–223. Springer, 2005.

[4] J. Corbett, M. B. Dwyer, J. Hatcliff, C. S. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera :

Extracting Finite-state Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering*, Limerich, Ireland, June 2000. ACM Press.

[5] M. Dahm. BCEL. `http://jakarta.apache.org/bcel`.

[6] D. L. Detlefs, K. Rustan M. Leino, G. Nelson, and J. B. Saxe. Extended Static Checking. Technical Report 159, Compaq Systems Research Center, Palo Alto, California, USA, 1998.

[7] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded Java Program Test Generation. *Software Testing and Verification*, 41(1), 2002.

[8] E. Farchi, Y. Nir-Buchbinder, and S. Ur. A Cross-Run Lock Discipline Checker for Java. Tool presented at the Parallel and Distributed Systems: Testing and Debugging (PADTAD) track of the 2005 IBM Verification Conference, Haifa, Israel. Tool is available at `http://alphaworks.ibm.com/tech/contest`, November 2005.

[9] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 174–186, Paris, France, January 1997.

[10] K. Havelund. Using Runtime Analysis to Guide Model Checking of Java Programs. In *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 245–264. Springer, 2000.

[11] K. Havelund and T. Pressburger. Model Checking Java Programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, April 2000. Special issue of STTT containing selected submissions to the 4th SPIN workshop, Paris, France, 1998.

[12] K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. *Proceedings of the First International Workshop on Runtime Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*, pages 97–114, Paris, France, July 2001. Elsevier Science.

[13] J. Harrow. Runtime Checking of Multithreaded Applications with Visual Threads. In *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 331–342. Springer, 2000.

[14] G. J. Holzmann and M. H. Smith. A Practical Method for Verifying Event-Driven Software. In *Proceedings of ICSE'99, International Conference on Software Engineering*, Los Angeles, California, USA, May 1999. IEEE/ACM.

[15] E. Knapp. Deadlock Detection in Distributed Database Systems. *ACM Computing Surveys*, pages 303–328, Dec. 1987.

[16] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads Programming.* O'Reilly, 1998.

[17] D. Park, U. Stern, J. Skakkebaek, and D. Dill. Java Model Checking. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, pages 253–256, September 2000.

[18] PolySpace. An Automatic Run-Time Error Detection Tool.
`http://www.polyspace.com`.

[19] M. Singhal. Deadlock Detection in Distributed Systems. *IEEE Computer*, pages 37–48, Nov. 1989.

[20] S. D. Stoller. Model-Checking Multi-threaded Distributed Java Programs. In *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 224–244. Springer, 2000.

[21] R. Agarwal, L. Wang, and S. D. Stoller. Detecting Potential Deadlocks with Static Analysis and Run-Time Monitoring. In *Proceedings of the Parallel and Distributed Systems: Testing and Debugging (PADTAD) track of the 2005 IBM Verification Conference, Haifa, Israel.* Springer-Verlag, November 2005. These proceedings.

[22] W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *Proceedings of ASE'00: The 15th IEEE International Conference on Automated Software Engineering.* IEEE CS Press, September 2000.