Scalable Dynamic Deadlock Analysis of Multi-Threaded Programs

Saddek Bensalem[1] and Klaus Havelund[2]

[1] Université Joseph Fourier, Verimag, Grenoble, France
[2] Kestrel Technology, Palo Alto, California USA

Abstract. This paper presents a dynamic program analysis algorithm that can detect deadlock potentials in a multi-threaded program by examining a single execution trace, obtained by running an instrumented version of the program. The algorithm is interesting because it can identify deadlock potentials even though no deadlocks occur in the examined execution, and therefore it scales very well in contrast to more formal approaches to deadlock detection. It is an improvement of an existing algorithm in that it reduces the number of false positives (false warnings). The paper describes an implementation, Java PathExplorer, for analyzing Java programs. An application of the implementation to two case studies is described.

1 Introduction

The Java programming language [1] explicitly supports concurrent programming through a selection of concurrency language concepts, such as threads and monitors. Threads execute in parallel, and communicate via shared objects that can be locked using synchronized access to achieve mutual exclusion. However, with concurrent programming comes a new set of problems that can hamper the quality of the software. Deadlocks form such a problem category. That deadlocks pose a common problem is emphasized by the following statement in [20]: "*Among the most central and subtle liveness failures is deadlock. Without care, just about any design using synchronization on multiple cooperating objects can contain the possibility of deadlock*".

In this paper we present a dynamic program analysis algorithm that can detect the potential for deadlocks in a program by analyzing a trace (log file) generated from a successful deadlock free execution of the program. The algorithm is interesting because it catches existing deadlock potentials with very high probability even when no actual deadlocks occur during test runs. A basic version of this algorithm has previously been implemented in the commercial tool Visual Threads [14]. This basic algorithm, however, can give false positives (as well as false negatives), putting a burden on the user to refute such. Our goal is to reduce the amount of false positives reported by the algorithm, and for that purpose we have modified it as reported in this paper. Detection of errors in concurrent programs by analysis of successful runs was first suggested for low-level data races in [23]. Other forms of data races have later shown to be detectable using related forms of analysis, such as high-level data races [3] and atomicity violations [4].

Two types of deadlocks have been discussed in the literature [24] [19]: *resource deadlocks* and *communication deadlocks*. In resource deadlocks, a process which requests resources must wait until it acquires all the requested resources before it can proceed with its computation. A set of processes is resource deadlocked if each process in the set requests a resource held by another process in the set, forming a *cycle* of lock requests. In communication deadlocks, messages are the resources for which processes wait. Reception of a message takes a process out of wait. A set of processes is communication deadlocked if each process in the set is waiting for a message from another process in the set and no process in the set ever sends a message. In this paper we focus only on *resource deadlocks*.

In Java, threads can communicate via shared objects by for example calling methods on those objects. In order to avoid data races in these situations (where several threads access a shared object simultaneously), objects can be locked using the `synchronized` statement, or by declaring methods on the shared objects `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 released when the scope of the `synchronized` statement is left; that is, when execution passes the curly bracket: '}'. Java also provides the `wait` and `notify` primitives in support for user controlled interleaving between threads. While the `synchronized` primitive is the main source for resource deadlocks in Java, the `wait` and `notify` primitives are the main source for communication deadlocks. Since this paper focuses on resource deadlocks, we shall in the following focus on Java's capability of creating and executing threads and on the `synchronized` statement.

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. This means in particular that generating the particular executions that expose a deadlock is difficult. Various technologies have been developed by the formal methods community to circumvent this problem, such as static analysis and model checking. Static analysis, such as performed by tools like JLint [2], PolySpace [22] and ESC [10], analyze the source code without executing it. These techniques are very efficient, but they often yield many false positives (false warnings) and additionally cannot well analyze programs where the object structure is very dynamic. Model checking has been applied directly to software (in contrast to only designs), for example in the Java PathFinder system (JPF) developed by NASA [16, 26], and in similar systems [12, 18, 9, 5, 25, 21, 11]. 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, especially for large realistic programs consisting of thousands of lines of code.

Static analysis and model checking are both typically complete (no false negatives), and model checking in addition is typically sound (no false positives). The algorithm presented in this paper 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. The technique is based on trace analysis: a program is instrumented to log synchronization events when executed. The algorithm then examines the log file, building a lock graph, which reveals deadlock potentials by containing *cycles*. The algorithm has been implemented in the Java PathExplorer tool [17], which in addition analyzes Java programs for various forms of data races [23, 3] and conformance with temporal logic properties [6]. Although the implementation is Java specific, the principles and theory presented are universal and apply in full to multi-threaded programs written in languages like C and C++ as well. The algorithm was first described in [7].

The paper is organized as follows. Section 2 introduces preliminary concepts and notation used throughout the rest of the paper. Section 3 introduces an example that illustrates the notion of deadlock and the different forms of false positives that are possible. Section 4 presents the basic algorithm suggested in [14] (the algorithm is only explained in few words in [14]). The subsequent two sections 5 and 6 suggest modifications, each reducing false positives. Section 7 shortly describes the implementation of the algorithm and presents the results of a couple of case studies. Finally, Section 8 concludes the paper.

2 Preliminaries

A directed graph is a pair $G = (S, R)$ of sets satisfying $R \subseteq S \times S$. The set R is called the edge set of G, and its elements are called edges. A path p is a non-empty graph $G = (S, R)$ of the form $S = \{x_1, x_2, \ldots, x_k\}$ and $R = \{(x_1, x_2), (x_2, x_3), \ldots, (x_{k-1}, x_k)\}$, where the x_i are all distinct, except that x_k may be equal to x_1, in which case the path is a cycle. The nodes x_0 and x_k are linked by p; we often refer to a path by the natural sequence of its nodes, writing, say, $p = x_1, x_2, \ldots, x_k$ and calling p a path from x_1 to x_k. In case where the edges are labeled with elements in L, G is triplet (S, L, R) and called a labeled graph with $R \subseteq S \times L \times S$. A labeled path, respectively cycle, is a labeled graph with the obvious meaning. Given a sequence $\sigma = x_1, x_2, \ldots, x_n$, we refer to an element at position i in σ by $\sigma[i]$ and the length of σ by $|\sigma|$. We let $<>$ denote the empty sequence, and the concatenation of two sequences σ_1 and σ_2 is denoted by $\sigma_1 \frown \sigma_2$. We denote by σ^i the prefix $x_1, \ldots, x_i$. Let $M : [A \xrightarrow{m} B]$ be a finite domain mapping from elements in A to elements in B (the $\xrightarrow{m}$ operator generates the set of finite domain mappings from A to B, hence partial functions on A). We let $M \dagger [a \mapsto b]$ denote the mapping M overwritten with a mapping to b. That is, the † operator represents map overwriting, and $[a \mapsto b]$ represents a map that maps a to b. Looking up the value mapped to by a in M is denoted by $M[a]$. We denote the empty mapping by $[\,]$.

3 An Example

We shall with an example illustrate the three categories of false positives that the basic algorithm reports, but which the improved algorithm will not report. The first category, *single threaded cycles*, refers 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. The program in Figure 1 illustrates these three situations, and a true positive. The real deadlock potential exists between threads T_2 and T_3, corresponding

$Main$:

```
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: }
```

Fig. 1. Example containing four lock cycles

to a cycle on L_1 and L_2. 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.

When analyzing such a program for deadlock potentials, we are interested in observing all lock acquisitions and releases, and all thread starts and joins. The program can be instrumented to emit such events. A lock trace $\sigma = e_1, e_2, \ldots, e_n$ is a finite sequence of lock and unlock events and start and join events. Let E_σ denote the set of events occurring in σ. Let T_σ denote the set of threads occurring in E_σ, and let L_σ denote the set of locks occurring in E_σ. We assume for convenience that the trace is *reentrant free* in the sense that an already acquired lock is never re-acquired by the same thread (or any other thread of course) before being released. Note that Java supports reentrant locks by allowing a lock to be re-taken by a thread that already has the lock. However, the instrumentation can generate reentrant free traces if it is recorded how many times a lock has been acquired nested by a thread. Normally a counter is introduced that is incremented for each lock operation and decremented for each unlock operation. A lock operation is now only reported if the counter is zero (it is free before being taken), and an unlock operation is only reported if the counter is 0 again after the unlock (it becomes free again).

For illustration purposes we shall assume a non-deadlocking execution trace σ for this program. It doesn't matter which one since all non-deadlocking traces will reveal all four cycles in the program using the basic algorithm. We shall assume the following trace of line numbered events (the line number is the first argument), which first, after having started T_1 and T_2 from the $Main$ thread, executes T_1 until the join statement, then executes T_2 to the end, then T_3 to the end, and then continues with T_1 after it has joined on T_3's termination. The line numbers are given for illustration purposes, and are actually recorded in the implementation in order to provide the user with useful error messages. In addition to the lock and unlock events $l(lno, t, o)$ and $u(lno, t, o)$ for line numbers lno, threads t and locks o, the trace also contains events for thread start, $s(lno, t_1, t_2)$ and thread join, $j(lno, t_1, t_2)$, meaning respectively that t_1 starts or joins t_2 in line number lno.

$\sigma =$
$s(1, Main, T_1),\ \ s(2, Main, T_2),$
$l(3, T_1, G),\ \ l(4, T_1, L_1),\ \ l(5, T_1, L_2),\ \ u(5, T_1, L_2),\ \ u(6, T_1, L_1),\ \ u(7, T_1, G),\ \ s(9, T_1, T_3),$
$l(14, T_2, G),\ \ l(15, T_2, L_2),\ \ l(16, T_2, L_1),\ \ u(16, T_2, L_1),\ \ u(17, T_2, L_2),\ \ u(18, T_2, G),$
$l(19, T_3, L_1),\ \ l(20, T_3, L_2),\ \ u(20, T_3, L_2),\ \ u(21, T_3, L_1),$
$j(10, T_1, T_3),\ \ l(11, T_1, L_2),\ \ l(12, T_1, L_1),\ \ u(12, T_1, L_1),\ \ u(13, T_1, L_2)$

Occasionally line numbers will be left out when referring to events.

4 Basic Cycle Detection Algorithm

In essence, the detection algorithm consists of finding cycles in a *lock graph*. In the context of multi-threaded programs, the basic algorithm sketched in [14] 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. Any cycle in the graph signifies a potential for a deadlock. Hence, we shall initially restrict ourselves to traces including only lock and unlock events (no start or join events). In order to define the lock graph, we introduce a notion that we call a *lock context* of a trace σ in position i, denoted by $\mathcal{C}_L(\sigma, i)$. It is a mapping from each thread to the set of locks owned by that thread at that position. Formally, for a thread $t \in T_\sigma$ we have the following: $\mathcal{C}_L(\sigma, i)(t) = \{o \mid \exists j : j \leq i \wedge \sigma[j] = l(t, o) \wedge \neg\exists k : j < k \leq i \wedge \sigma[k] = u(t, o)\}$. Bellow we give a definition that allows to build the lock graph G_L with respect to an execution trace σ. An edge in G_L between two locks l_1 and l_2 means that there exists a thread t which owns the object l_1 while taking the object l_2.

Definition 1 (Lock graph) *Given an execution trace $\sigma = e_1, e_2, \ldots, e_n$. We say that the lock graph of σ is the minimal directed graph $G_L = (L, R)$ such that : L is the set of locks L_σ, and $R \subseteq L \times L$ is defined by $(l_1, l_2) \in R$ if there exists a thread $t \in T_\sigma$ and a position $i \geq 2$ in σ s. t. $\sigma[i] = l(t, l_2)$ and $l_1 \in \mathcal{C}_L(\sigma, i-1)(t)$.*

In Figure 2 we give an algorithm for constructing the lock graph from a lock trace. In this algorithm, we also use the context $\mathcal{C}_L$ which is exactly the same as in the definition 1. The only difference is that we don't need to explicitly use the two parameters σ and i. The set of cycles in the graph G_L, denoted by $cylces(G_L)$, represents the potential deadlock situations in the program. The lock graph for the example in Figure 1 is also shown in Figure 2.

```
**Input**: An execution trace $\sigma$
$G_L$ is a graph;
$\mathcal{C}_L : [T_\sigma \to 2^{L_\sigma}]$ is a lock context;
**for**($i = 1 .. |\sigma|$) **do**
    **case** $\sigma[i]$ **of**
        $l(t, o) \ \to$
            $G_L := G_L \bigcup \{(o', o) \mid o' \in \mathcal{C}_L(t)\}$;
            $\mathcal{C}_L := \mathcal{C}_L \dagger [t \mapsto \mathcal{C}_L(t) \bigcup \{o\}]$;
        $u(t, o) \ \to$
            $\mathcal{C}_L := \mathcal{C}_L \dagger [t \mapsto \mathcal{C}_L(t) \backslash \{o\}]$
**end**;
**for each** c in cycles($G_L$) **do**
    **print** ("deadlock potential:",c);
```

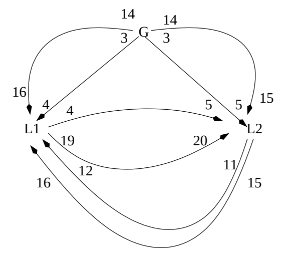

Fig. 2. The basic algorithm and the lock graph

5 Eliminating Single Threaded and Guarded Cycles

In this section we present a solution that removes false positives stemming from *single threaded cycles* and *guarded cycles*. In [15] we suggested a solution, the GoodLock algorithm, based on building synchronization trees. However, this solution could only detect deadlocks between pairs of threads. The algorithm to be presented here is not limited in this sense. The solution is to extend the lock graph by labeling each edge between locks with information about which thread causes the addition of the edge and what gate locks were held by that thread when the target lock was taken. A definition of *valid cycles* will then include this information to filter out false positives. First, we define the extended lock graph.

Definition 2 (Guarded lock graph) *Given a trace $\sigma = e_1, e_2, \ldots, e_n$. We say that the guarded lock graph of σ is the minimal directed labeled graph $G_L =$*

(L, W, R) such that: L is the set of locks L_σ, $W \subseteq T_\sigma \times 2^L$ is the set of labels, each containing a thread id and a lock set, and $R \subseteq L \times W \times L$ is defined by $(l_1, (t, g), l_2) \in R$ if there exists a thread $t \in T_\sigma$ and a position $i \geq 2$ in σ s.t. $\sigma[i] = l(t, l_2)$ and $l_1 \in \mathcal{C}(\sigma, i-1)(t)$ and $g = \mathcal{C}(\sigma, i-1)(t)$.

Each edge $(l_1, (t, g), l_2)$ in R is labeled with the thread t that took the locks l_1 and l_2, and a lock set g, indicating what locks t owned when taking l_2. In order for a cycle to be valid, and hence regarded as a true positive, the threads and guard sets occurring in labels of the cycle must be valid in the following sense.

Definition 3 (Valid threads and guards) *Let G_L be a guarded lock graph of some execution trace and $c = (L, W, R)$ a cycle in cycles(G_L), we say that:*

- *threads of c are valid if: forall $e, e' \in W$ $e \neq e' \Rightarrow thread(e) \neq thread(e')$*
- *guards of c are valid if: forall $e, e' \in W$ $e \neq e' \Rightarrow guards(e) \cap guards(e') = \emptyset$*

where, for a label $e \in W$, $thread(e)$, resp. $guards(e)$, gives the first, resp. second, component of e.

For a cycle to be valid, the threads involved must differ. This eliminates single threaded cycles. Furthermore, the lock sets on the edges in the cycle must not overlap. This eliminates cycles that are guarded by the same lock taken "higher up" by at least two of the threads involved in the cycle. Assume namely that such a gate lock exists, then it will belong to the lock sets of several edges in the cycle, and hence they will overlap at least on this lock. This corresponds to the fact that a deadlock cannot happen in this situation. Valid cycles are now defined as follows:

Definition 4 (Unguarded cycles) *Let σ be an execution trace and G_L its guarded lock graph. We say that a cycle $c \in cycles(G_L)$ is an unguarded cycle if the guards of c are valid and threads of c are also valid. We denote by $cycles_g(G_L)$ the set of unguarded cycles in $cycles(G_L)$.*

We shall in this section not present an explicit algorithm for constructing this graph, since its concerns a relatively simple modification to the basic algorithm – the statement that updates the lock graph becomes:

$$G_L := G_L \bigcup \{(o', (t, \mathcal{C}(t)), o) \mid o' \in \mathcal{C}(t)\}$$

adding the labels $(t, \mathcal{C}(t))$ to the edges. Furthermore, cycles to be reported should be drawn from: $cycles_g(G_L)$.

Let us illustrate the algorithm with an example. We consider again the execution trace σ from Section 3. The guarded graph for this trace is shown in Figure 3. The graph contains the same number of edges as the basic graph in Figure 2.

However, now edges are labeled with a thread and a guard set. In particular, we notice that the gate lock G occurs in the guard set of edges $(4, 5)$ and $(15, 16)$. This prevents this guarded cycle from being included in the set of valid cycles since it is not guard valid: the guard sets overlap in G. Also the single threaded cycle $(4, 5) \leftrightarrow (11, 12)$ is eliminated because it is not thread valid: the same thread T_1 occurs on both edges.

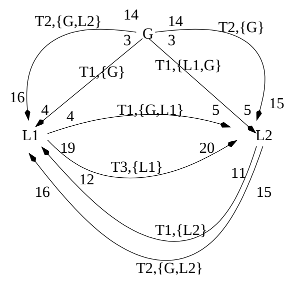

Fig. 3. Guarded lock graph

6 Eliminating Segmented Cycles

In the previous section we saw the specification of an algorithm that removes false positives stemming from single threaded cycles and guarded cycles. In this section we present the full algorithm that in addition removes false positives stemming from *segmented cycles*. We assume that traces now also contain start and join events. Recall the example in Figure 1 and that the basic algorithm reports a cycle between threads T_1 (line 11-12) and T_3 (line 19-20) on locks L_1 and L_2. However, a deadlock is impossible since thread T_3 is joined on by T_1 in line 10. Hence, the two *code segments*: line 11-12 and line 19-20 can never run in parallel. The algorithm to be presented will prevent such cycles from being reported by formally introducing such a notion of *segments* that cannot execute in parallel. A new directed segmentation graph will record which segments execute before others. The lock graph is then extended with extra label information, that specifies what segments locks are acquired in, and the validity of a cycle now incorporates a check that the lock acquisitions are really occurring in parallel executing segments. The idea of using segmentation in runtime analysis was initially suggested in [14] to reduce the amount of false positives in data race analysis using the Eraser algorithm [23]. We use it in a similar manner here to reduce false positives in deadlock detection.

More specifically, the solution is during execution to associate segment identifiers (natural numbers, starting from 0) to segments of the code that are separated by statements that *start* or *join* other threads. For example, if a thread t_1 currently

is in segment s and starts another thread t_2, and the next free segment is $n+1$, then t_1 will continue in segment $n+1$ and t_2 will start in segment $n+2$. From then on the next free segment will be $n+3$. It is furthermore recorded in the segmentation graph that segment s executes before $n+1$ as well as before $n+2$. In a similar way, if a thread t_1 currently is in segment s_1 and joins another thread t_2 that is in segment s_2, and the next free segment is $n+1$, then t_1 will continue in segment $n+1$, t_2 will be terminated, and from then on the next free segment will be $n+2$. It is recorded that s_1 as well as s_2 execute before $n+1$. Figure 5 illustrates the segmentation graph for the program example in Figure 1. In order to give a formal definition of the segmentation we need to define two functions. The first one, $\mathcal{C}_S(\sigma)$, *segmentation context* of the trace σ, gives for each position i of the execution trace σ, the current segment of each thread t at that position. Formally, $\mathcal{C}_S(\sigma)$ is the mapping with type: $[\mathcal{N} \mapsto [T_\sigma \mapsto \mathcal{N}]]$, associated to trace σ, that maps each position into another mapping that maps each thread id to its current segment in that position. It is defined as follows. Let $\mathcal{C}_S^{init} = [0 \mapsto [main \mapsto 0]]$, mapping position 0 to the mapping that maps the main thread to segment 0. Then $\mathcal{C}_S(\sigma)$ is defined by the use of the auxiliary function $f_0 : Trace \times Context \times Position \times Current\_Segment \to Context$:
$\mathcal{C}_S(\sigma) = f_0(\sigma, \mathcal{C}_S^{init}, 1, 0)$, where the function f_0 is defined by left-to-right recursion over the trace σ as follows:

$$f_0(e \frown \sigma, C_S, i, n) = \begin{cases} f_0(\sigma, C_S, i+1, n) \\ \quad \textbf{if } \mathrm{e} \in \{\mathrm{l(t,o),u(t,o)}\}, \\ \\ f_0(\sigma, C_S \dagger [i \mapsto C_S[i-1] \dagger \begin{bmatrix} t_1 \mapsto n+1 \\ t_2 \mapsto n+2 \end{bmatrix}, i+1, n+2) \\ \quad \textbf{if } e = s(t_1, t_2), \\ \\ f_0(\sigma, C_S \dagger [i \mapsto C_S[i-1] \dagger [t_1 \mapsto n+1], i+1, n+1) \\ \quad \textbf{if } e = j(t_1, t_2). \end{cases}$$

$$f_0(<>, C_S, i, n) = C_S$$

The second function needed, $\#_{alloc}$, gives the number of segments allocated in position i of σ. This function is used to calculate what is the next segment to be assigned to a new execution block, and is dependent on the number of start events $s(t_1, t_2)$ and join events $j(t_1, t_2)$ that occur in the trace up and until position i, recalling that each start event causes two new segments to be allocated. Formally we define it as follows : $\#_{alloc}(\sigma, i) = |\sigma^i \downarrow_s| * 2 + |\sigma^i \downarrow_j|$.

We can now define the notion of a directed *segmentation graph*, which defines an ordering between segments. Informally, assume that in trace position i a thread t_1, being in segment $s_1 = \mathcal{C}_S(\sigma)(i-1)(t_1)$, executes a start of a thread t_2. Then t_1 continues in segment $n = \#_{alloc}(\sigma, i-1) + 1$ and t_2 continues in segment $n+1$. Consequently, (s_1, n) as well as $(s_1, n+1)$ belongs to the graph, meaning that s_1 executes before n as well as before $n+1$. Similarly, assume that a thread

t_1 in position i, being in segment $s_1 = \mathcal{C}_S(\sigma)(i-1)(t_1)$, executes a join of a thread t_2, being in segment $s_2 = \mathcal{C}_S(\sigma)(i-1)(t_2)$. Then t_1 continues in segment $n = \#_{alloc}(\sigma, i-1)+1$ while t_2 terminates. Consequently (s_1, n) as well as (s_2, n) belongs to the graph, meaning that s_1 as well as s_2 executes before n. The formal definition of the segmentation graph is as follows.

Definition 5 (Segmentation graph) *Given an execution trace $\sigma = e_1, \ldots, e_n$. We say that a segmentation graph of σ is the directed graph $G_S = (\mathcal{N}, R)$ where: $\mathcal{N} = \{n \mid 0 \leq n \leq \#_{alloc}(\sigma, |\sigma|)\}$ is the set of segments, and $R \subseteq \mathcal{N} \times \mathcal{N}$ is the relation given by $(s_1, s_2) \in R$ if there exists a position $i \geq 1$ s.t. $\sigma[i] = s(t_1, t_2) \wedge s_1 = \mathcal{C}_S(\sigma)(i-1)(t_1) \wedge (s_2 = \#_{alloc}(\sigma, i-1)+1 \vee s_2 = \#_{alloc}(\sigma, i-1)+2)$, or $\sigma[i] = j(t_1, t_2) \wedge (s_1 = \mathcal{C}_S(\sigma)(i-1)(t_1) \vee s_1 = \mathcal{C}_S(\sigma)(i-1)(t_2)) \wedge s_2 = \#_{alloc}(\sigma, i-1)+1$.*

The following relation *happens-before* reflects how the segments are related in time during execution.

Definition 6 (Happens-Before relation) *Let $G_S = (\mathcal{N}, R)$ be a segmentation graph, and $G_S^* = (\mathcal{N}, R^*)$ its transitive closure. Then given two segments s_1 and s_2, we say that s_1 happens before s_2, denoted by $s_1 \rhd s_2$, if $(s_1, s_2) \in R^*$.*

Note that for two given segments s_1 and s_2, if neither $s_1 \rhd s_2$ nor $s_2 \rhd s_1$, then we say that s_1 *happens in parallel* with s_2. Before we can finally define what is a lock graph with segment information, we need to redefine the notion of lock context, $\mathcal{C}_L(\sigma, i)$, of a trace σ and a position i, that was defined on page 5. In the previous definition it was a mapping from each thread to the set of locks owned by that thread at that position. Now we add information about what segment each lock was taken in. Formally, for a thread $t \in T_\sigma$ we have the following :

$$\begin{aligned}&\mathcal{C}_L(\sigma, i)(t) = \\ &\quad\{(o, s) \mid \exists j : j \leq i \ \wedge \ \sigma[j] = l(t, o) \ \wedge \\ &\qquad\quad \neg(\exists k : j < k \leq i \wedge \sigma[k] = u(t, o)) \ \wedge \ \mathcal{C}_S(\sigma)(j)(t) = s\}\end{aligned}$$

An edge in G_L between two locks l_1 and l_2 means, as before, that there exists a thread t which owns an object l_1 while taking the object l_2. The edge is as before labeled with t as well as the set of (gate) locks. In addition, the edge is now further labeled with the segments s_1 and s_2 in which the locks l_1 and l_2 were taken by t.

Definition 7 (Segmented and guarded lock graph) *Given an execution trace $\sigma = e_1, e_2, \ldots, e_n$. We say that the segmented and guarded lock graph of σ is the minimal directed graph $G_L = (L_\sigma, W, R)$ such that:*

- *$W \subseteq \mathcal{N} \times (T_\sigma \times 2^{L_\sigma}) \times \mathcal{N}$ is the set of labels $(s_1, (t, g), s_2)$, each containing the segment s_1 that the source lock was taken in, a thread id t, a lock set g (these two being the labels of the guarded lock graph in the previous section), and the segment s_2 that the target lock was taken in,*

– *$R \subseteq L_\sigma \times W \times L_\sigma$ is defined by $(l_1, (s_1, (t, g), s_2), l_2) \in R$ if there exists a thread $t \in T_\sigma$ and a position $i \geq 2$ in σ such that: $\sigma[i] = l(t, l_2)$ and $(l_1, s_1) \in \mathcal{C}_L(\sigma)(i-1)(t)$ and $g = \{l' \mid (l', s) \in \mathcal{C}_L(\sigma)(i-1)(t)\}$ and $s_2 = \mathcal{C}_S(\sigma)(i-1)(t)$*

Each edge $(l_1, (s_1, (t, g), s_2), l_2)$ in R is labeled with the thread t that took the locks l_1 and l_2, and a lock set g, indicating what locks t owned when taking l_2. The segments s_1 and s_2 indicate in which segments respectively l_1 and l_2 were taken.

In order for a cycle to be valid, and hence regarded as a true positive, the threads and guard sets occurring in labels of the cycle must be valid as before. In addition, the segments in which locks are taken must now allow for a deadlock to actually happen. Consider for example a cycle between two threads t_1 and t_2 on two locks l_1 and l_2. Assume further that t_1 takes l_1 in segment x_1 and then l_2 in segment x_2 while t_2 takes them in opposite order, in segments y_1 and y_2 respectively. Then it must be possible for t_1 and t_2 to each take their first lock in order for a deadlock to occur. In other words, x_2 must not happen before y_1 and y_2 must not happen before x_1. This is expressed in the following definition, which repeats the definitions from Definition 3.

Definition 8 (Valid threads, guards and segments) *Let G_L be a segmented and guarded lock graph of some execution trace and $c = (L, W, R)$ a cycle in $cycles(G_L)$, we say that:*

– *threads of c are valid if: forall $e, e' \in W$, $e \neq e' \Rightarrow thread(e) \neq thread(e')$*
– *guards of c are valid if: forall $e, e' \in W$, $e \neq e' \Rightarrow guards(e) \cap guards(e') = \emptyset$*
– *segments of c are valid if: forall $e, e' \in W$, $e \neq e' \Rightarrow \neg\,(seg_2(e_1) \triangleright seg_1(e_2))$*

where, for a label $e = (s_1, (t, g), s_2) \in W$, $thread(e) = t$, $guards(e) = g$, $seg_1(e) = s_1$ and $seg_2(e) = s_2$.

Valid cycles are now defined as follows.

Definition 9 (Unsegmented and unguarded cycles) *Let σ be an execution trace and G_L its segmented and guarded lock graph. We say that a cycle $c \in cycles(G_L)$ is an unsegmented and unguarded cycle if the guards of c are valid, the threads of c are valid, and the segments of c are valid. We denote by $cycles_s(G_L)$ the set of unsegmented and unguarded cycles in $cycles(G_L)$.*

Figure 4 presents an algorithm for constructing the segmentation graph and lock graph from an execution trace. The set of cycles in the graph G_L, denoted by $cylces_s(G_L)$, see Definition 9, represents the potential deadlock situations in the program. The segmentation graph (G_S) and lock graph (G_L) have the structure as outlined in Definition 5 and Definition 7 respectively. The lock context (C_L) maps each thread to the set of locks owned by that thread at any point in time. Associated with each such lock is the segment in which it was acquired. The segment context (C_S) maps each thread to the segment in which it is currently executing. The algorithm should after this explanation and the previously given abstract definitions be self explanatory. Consider again the trace

```
**Input**: An execution trace $\sigma$
$G_L$ is a lock graph;
$G_S$ is a segmentation graph;
$C_L : [T_\sigma \rightarrow 2^{L_\sigma \times \mathbf{nat}}]$ is a lock context;
$C_S : [T_\sigma \rightarrow \mathbf{nat}]$ is a segment context;
$n : \mathbf{nat} = 1$ next available segment;
for($i = 1 .. |\sigma|$) do
    case $\sigma[i]$ of
        $l(t, o) \rightarrow$
            $G_L := G_L \bigcup \{(o', (s_1, (t, g), s_2), o) \mid$
                                  $(o', s_1) \in C_L(t) \wedge$
                                  $g = \{o'' \mid (o'', s) \in C_L(t)\} \wedge$
                                  $s_2 = C_S(t)\}$;
            $C_L := C_L \dagger [t \mapsto C_L(t) \bigcup \{(o, C_S(t))\}]$;
        $u(t, o) \rightarrow$
            $C_L := C_L \dagger [t \mapsto C_L(t) \backslash \{(o, *)\}]$;
        $s(t_1, t_2) \rightarrow$
            $G_S := G_S \bigcup \{(C_S(t_1), n), (C_S(t_1), n + 1)\}$;
            $C_S := C_S \dagger [t_1 \mapsto n, t_2 \mapsto n + 1]$;
            $n := n + 2$;
        $j(t_1, t_2) \rightarrow$
            $G_S := G_S \bigcup \{(C_S(t_1), n), (C_S(t_2), n)\}$;
            $C_S := C_S \dagger [t_1 \mapsto n]$;
            $n := n + 1$;
end;
for each $c$ in $cycles_s(G_L)$ do
    print ("deadlock potential:",$c$);
```

Fig. 4. The final algorithm

σ from Section 3. The segmented and guarded lock graph and the segmentation graph for this trace are both shown in Figure 5. The segmentation graph is for illustrative purposes augmented with the statements that caused the graph to be updated. We see in particular that segment 6 of thread T_3 executes before segment 7 of thread T_1, written as $6 \rhd 7$. Segment 6 is the one in which T_3 executes lines 19 and 20, while segment 7 is the one in which T_1 executes lines 11 and 12. The lock graph contains the same number of edges as the guarded graph in Figure 3, and the same *(thread,guard set)* labels. However, now edges are additionally labeled with the segments in which locks are taken. This makes the cycle $(19, 20) \leftrightarrow (11, 12)$ segment invalid since the target segment of the first edge (6) executes before the source segment of the second edge (7).

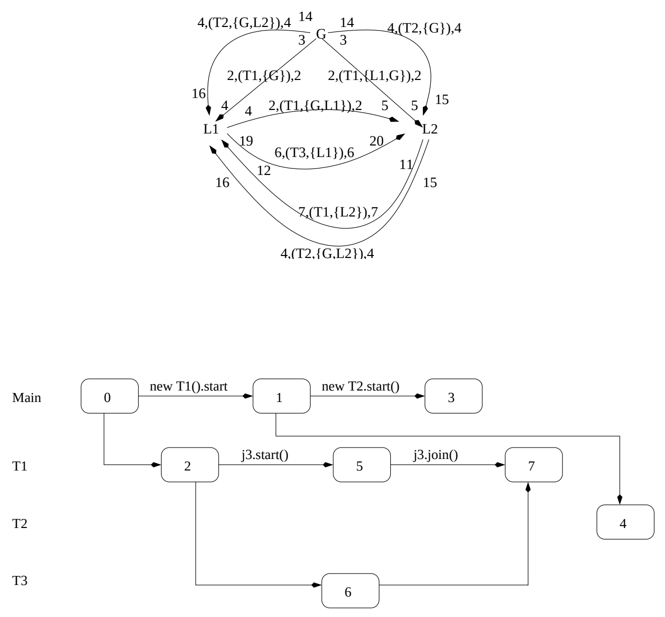

Fig. 5. Segmented lock graph (above) and segmentation graph (below)

7 Implementation and Experimentation

The algorithm presented in the previous section has been implemented in the Java PathExplorer (JPaX) tool [17]. JPaX consists of two main modules, an *instrumentation module* and an *observer module*. The instrumentation module automatically instruments the bytecode class files of a compiled program by adding new instructions that when executed generate the execution trace consisting of the events needed for the analysis. The observer module reads the event stream and dispatches this to a set of observer rules, each rule performing a particular analysis that has been requested, such as deadlock analysis, data race analysis and temporal logic analysis. This modular rule based design allows a user to easily implement new runtime verification algorithms. The Java bytecode instrumentation is performed using the jSpy instrumentation package [13] that is part of Java PathExplorer. jSpy's input is an instrumentation specification, which consists of a collection of rules, where a rule is a predicate/action pair. The predicate is a conjunction of syntactic constraints on a Java statement, and the action is a description of logging information to be inserted in the bytecode corresponding to the statement. As already mentioned, this form of analysis is not complete and hence may yield false negatives by missing to report synchronization problems. A synchronization problem can most obviously be missed if one or more of the synchronization statements involved in the problem do not get executed. To avoid being entirely in the dark in these situations, we added a coverage module to the system that records what lock-related instructions are instrumented and which of these that are actually executed.

JPaX's deadlock analyzer has been applied to two NASA case studies, a planetary rover controller named K9, originally programmed in C++ and later translated to Java as part of an attempt to evaluate verification tools and Java for mission software; and a planner named Europa programmed in C++. In the latter case ASCII log files were generated which was analyzed by the tool. The Java version of K9 was in particular created to evaluate a range of program verification technologies, among them JPaX, as described in [8]; the other technologies included static analysis and model checking. Two resource deadlocks were seeded in K9 by an independent team. JPaX immediately found both deadlocks and generally came out well in the comparison, as being fast and effective. In addition, an early version of the deadlock algorithm found a deadlock in the original C++ version of K9 that was unexpected by the programmer. This experiment was performed by creating a C++ specific instrumentation module, whereas the observer module could be used unmodified. In the planner Europa, the tool found 2 deadlock potentials that were unknown to the programming team. This result caused the team to request an integration of the observer part into their development suite for future use.

8 Conclusions

An algorithm has been presented for detecting deadlock potentials in concurrent programs by analyzing execution traces. The algorithm extends an existing algorithm by reducing the amount of false positives reported, and has been implemented in the JPaX tool. Although JPaX analyzes Java programs, it can be applied to applications written in other languages by modifying the instrumentation module. The advantage of trace analysis is that it scales extremely well, in contrast to more formal methods, and in addition can detect errors that for example static analysis cannot properly detect. In current work, we further approach the problem of false positives by developing a framework for generating test cases from warnings issued by this tool. Such test cases will then directly expose the deadlocks. Also, static analysis of deadlocks can be combined with dynamic analysis. Additional current work attempts to extend the capabilities of JPaX with new algorithms for detecting other kinds of concurrency errors, such as various forms of data races and communication deadlocks. An additional important issue is the performance impact on the instrumented program.

References

1. K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.
2. 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.
3. C. Artho, K. Havelund, and A. Biere. High-level Data Races. In *VVEIS '03*, April 2003. Extended version to appear in the journal *Software Testing, Verification and Reliability*.
4. C. Artho, K. Havelund, and A. Biere. Using Block-Local Atomicity to Detect Stale-Value Concurrency Errors. In *2nd International Symposium on Automated Technology for Verification and Analysis, Taiwan*, October–November 2004.
5. 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.
6. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-Based Runtime Verification. In *Proceedings of Fifth International Conference on Verification, Model Checking and Abstract Interpretation, January 2004 – to appear*, August 2003.
7. S. Bensalem and K. Havelund. Reducing False Positives in Runtime Analysis of Deadlocks. Internal report, NASA Ames Research Center, October 2002.
8. G. Brat, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, A. Venet, and W. Visser. Experimental Evaluation of Verification and Validation Tools on Martian Rover Software. In *SEI Software Model Checking Workshop*, 2003. Extended version to appear in the journal *Formal Methods in System Design*.
9. 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.
10. 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.
11. O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded Java Program Test Generation. *Software Testing and Verification*, 41(1), 2002.
12. 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.
13. A. Goldberg and K. Havelund. Instrumentation of Java Bytecode for Runtime Analysis. In *Proc. Formal Techniques for Java-like Programs*, volume 408 of *Technical Reports from ETH Zurich*, Switzerland, 2003. ETH Zurich.
14. 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.
15. K. Havelund. Using Runtime Analysis to Guide Model Checking of Java Programs. In K. Havelund, J. Penix, and W. Visser, editors, *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 245–264. Springer, 2000.
16. 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.

17. K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In K. Havelund and G. Roşu, editors, *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. Extended version to appear in the journal *Formal Methods in System Design*.
18. 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.
19. E. Knapp. Deadlock Detection in Distributed Database Systems. *ACM Computing Surveys*, pages 303–328, Dec. 1987.
20. D. Lea. *Concurrent Programming in Java, Design Principles and Patterns.* Addison-Wesley, 1997.
21. 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.
22. PolySpace. An Automatic Run-Time Error Detection Tool. `http://www.polyspace.com`.
23. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.
24. M. Singhal. Deadlock detection in distributed systems. *IEEE Computer*, pages 37–48, Nov. 1989.
25. 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.
26. 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.