Detecting Potential Deadlocks with Static Analysis and Run-Time Monitoring*

Rahul Agarwal, Liqiang Wang, and Scott D. Stoller

Computer Science Dept., SUNY at Stony Brook, Stony Brook, NY 11794-4400
{ragarwal, liqiang, stoller}@cs.sunysb.edu
http://www.cs.sunysb.edu/~{ragarwal, liqiang, stoller}

Abstract. Concurrent programs are notorious for containing errors that are difficult to reproduce and diagnose. A common kind of concurrency error is deadlock, which occurs when a set of threads is blocked each trying to acquire a lock held by another thread in that set. Static and dynamic (run-time) analysis techniques exist to detect deadlocks.

Havelund's GoodLock algorithm detects potential deadlocks at run-time. However, it detects only potential deadlocks involving exactly two threads. This paper presents a generalized version of the GoodLock algorithm that detects potential deadlocks involving any number of threads. Run-time checking may miss errors in unexecuted code. On the positive side, run-time checking generally produces fewer false alarms than static analysis.

This paper explores the use of static analysis to automatically reduce the overhead of run-time checking. We extend our type system, Extended Parameterized Atomic Java (EPAJ), which ensures absence of races and atomicity violations, with Boyapati *et al.*'s deadlock types. We give an algorithm that infers deadlock types for a given program and an algorithm that determines, based on the result of type inference, which run-time checks can safely be omitted. The new type system, called Deadlock-Free EPAJ (DEPAJ), has the added benefit of giving stronger atomicity guarantees than previous atomicity type systems.

1 Introduction

Concurrent programs are notorious for containing errors that are difficult to reproduce and diagnose at run-time. Some common kind of programming errors include deadlocks, data races and atomicity violations. A *deadlock* occurs when each thread is blocked trying to acquire a lock held by another thread. A *data race* occurs when two threads concurrently access a shared variable and at least one of the accesses is a write. Atomicity is a common higher-level correctness requirement that expresses non-interference between concurrently executed methods. A method is *atomic* if every execution of the program is equivalent to an execution in which that method is executed without being interleaved with other concurrently executed methods. This paper focuses on detecting deadlocks.

* This work was supported in part by NSF under Grant CCR-0205376 and CNS-0509230 and ONR under Grants N00014-02-1-0363 and N00014-04-1-0722.

S. Ur, E. Bin, and Y. Wolfsthal (Eds.): Haifa Verification Conf. 2005, LNCS 3875, pp. 191–207, 2006.

The GoodLock algorithm [Hav00] detects potential deadlocks at run-time. However, it detects only potential deadlocks involving two threads, *i.e.*, each of those threads is blocked trying to acquire a lock held by the other thread. This paper presents a generalized version of GoodLock algorithm, that detects potential deadlocks involving any number of threads in other executions of the program.

Static analysis can also detect potential deadlocks. Boyapati, Lee and Rinard [BLR02] introduce a type system that ensures Java programs are deadlock-free. That type system extends Boyapati and Rinard's Parameterized Race Free Java (PRFJ) type system [BR01], which ensures Java programs are race-free.

Run-time checking and static analysis are both useful. Static analysis can guarantee that all executions of a program are deadlock-free; run-time checking cannot. However, due to limitations of the type system, some deadlock-free programs are not typable; the resulting warnings from the typechecker are called *false alarms*, and they may be difficult to diagnose. On the other hand, run-time checking generally produces fewer false alarms than static analysis; this is a significant practical advantage, since diagnosing all of the warnings from static analysis of large codebases may be expensive.

This paper extends our type system, Extended Parameterized Atomic Java (EPAJ) [SAWS05], which ensures absence of races and atomicity violations, with the deadlock types described in [BLR02]. The new type system, called Deadlock-Free EPAJ (DEPAJ), ensures absence of deadlocks due to locks and, as an added benefit, gives stronger atomicity guarantees than EPAJ and Atomic Java [FQ03], which do not consider deadlocks and hence may classify a method as atomic even if it could deadlock in the middle—something that cannot happen if the method executes without interruption by other threads.

The type systems and run-time analysis algorithms considered in this paper only attempt to detect potential deadlocks caused by locks. They do not consider wait/notify or other forms of condition synchronization and hence do not detect deadlocks due to them.

Manually annotating code with the necessary type annotations can be a significant burden, especially for legacy code. Type inference reduces the annotation burden by automatically determining types for some or all parts of the program. This paper presents a type inference algorithm for [BLR02]'s basic deadlock types.

Static analysis can be used to decrease the overhead of run-time checking, in the following way. First, our type inference algorithm infers deadlock types for the program. Run-time deadlock detection is then focused on fragments of code which were not typable. The user can inspect the run-time warnings, which are more likely to indicate real errors and can provide more detailed and specific diagnostic information; then, if desired, the user can inspect warnings from the typechecker. The goal is to reduce the overhead of run-time checking to a level where it can be used unobtrusively throughout the testing process, or even in deployed systems, instead of only during a limited period of testing focused on concurrency errors.

The rest of the paper is organized as follows. Sections 2, 3, 4 and 5 describe run-time detection of potential deadlocks, deadlock types, type inference for deadlock types, and our type system DEPAJ respectively. Section 6 presents our techniques for focused run-time detection of potential deadlocks. Section 7 presents our experiments, Section 8 discusses related work.

2 Run-Time Detection of Potential Deadlocks

The GoodLock algorithm [Hav00] detects potential deadlocks at run-time. It records a run-time lock tree for each thread. The run-time lock tree for a thread represents the nested pattern in which locks are acquired by the thread. Each node of the run-time lock tree is labeled with a lock and represents the thread acquiring that lock. There is an edge from a node n_1 to a node n_2 if n_1 represents the most recently acquired lock that the thread holds when it acquires the lock associated with n_2. At each instant, each run-time lock tree has one node designated as the *current node*; the path from the root of the tree to that node represents the nested acquires of locks held by that thread at that instant. If a thread re-acquires a lock that it already holds, its run-time lock tree does not contain a node representing the re-acquire. When a thread acquires a lock that it does not already hold, if there is already a child of the current node labeled with that lock, that child becomes the current node, otherwise a new child labeled with that lock is created and becomes the current node. At the end of the execution of the program, if there exist threads t_1 and t_2 and locks l_1 and l_2 such that t_1 acquires l_2 while holding l_1, and t_2 acquires l_1 while holding l_2, then a warning of potential deadlock is issued, unless there is a common lock, called a gate lock, that is held by both threads when they acquire l_1 and l_2; the gate lock prevents the acquires of l_1 and l_2 from being interleaved in a way that leads to deadlock. The worst-case time complexity of the algorithm is $O(|T|^3 \times |Thread|^2)$, where $|T|$ is the size of the largest run-time lock tree, and $Thread$ is the set of threads. However, this algorithm only detects potential deadlocks caused by interleaving of lock acquires in two threads.

We present a generalized version of the GoodLock algorithm that detects potential deadlocks involving any number of threads. In particular, it checks whether there exist distinct threads $t_0, \ldots, t_{m-1}$ and locks $l_0, \ldots, l_{m-1}$ such that, for all $i = 0..m-1$, t_i holds lock l_i while acquiring lock $l_{i+1 \ mod \ m}$. Note that we always ignore a thread re-acquiring a lock it already holds, so a thread acquiring $l_{i+1 \ mod \ m}$ while holding l_i implies $l_{i+1 \ mod \ m}$ and l_i are different locks. In the absence of other constraints on the schedule (*e.g.*, due to gate locks or start-join synchronization), such acquires can be interleaved in a way that leads to deadlock. We call this the Potential for Deadlock from Locks Ignoring GateLocks (PDL-IGL) condition.

The algorithm constructs a run-time lock tree for each thread during execution, as described above. At the end of the execution, it constructs a run-time lock graph, which is a directed graph $G = (V, E)$, where V contains all the nodes of all the run-time lock trees, and the set E of directed edges contains (1)

tree edges: the directed (from parent to child) edges in each of the run-time lock trees, and (2) *inter edges:* bidirectional edges between nodes that are labeled with the same lock and that are in different run-time lock trees.

For a run-time lock graph G, a *valid path* is a path that does not contain consecutive inter edges and such that nodes from each lock tree appear as at most one consecutive subsequence in the path. Similarly, a *valid cycle* is a cycle that does not contain consecutive inter edges and nodes from each thread appear as at most one consecutive subsequence in the cycle.

As an example, Figure 1 shows the run-time lock graph for the illustrative program in Figure 2. The graph in Figure 1 contains several cycles including the following three, where $\texttt{l}i^{\mathrm{T}j}$ denotes the node for lock $\texttt{l}i$ in the run-time lock tree for thread j: $\texttt{l3}^{\mathrm{T1}} \rightarrow \texttt{l3}^{\mathrm{T2}} \rightarrow \texttt{l3}^{\mathrm{T4}} \rightarrow \texttt{l3}^{\mathrm{T1}}$, $\texttt{l1}^{\mathrm{T1}} \rightarrow \texttt{l2}^{\mathrm{T1}} \rightarrow \texttt{l2}^{\mathrm{T2}} \rightarrow \texttt{l3}^{\mathrm{T2}} \rightarrow \texttt{l3}^{\mathrm{T1}} \rightarrow \texttt{l4}^{\mathrm{T1}} \rightarrow \texttt{l4}^{\mathrm{T3}} \rightarrow \texttt{l1}^{\mathrm{T3}} \rightarrow \texttt{l1}^{\mathrm{T1}}$, and $\texttt{l3}^{\mathrm{T1}} \rightarrow \texttt{l4}^{\mathrm{T1}} \rightarrow \texttt{l4}^{\mathrm{T4}} \rightarrow \texttt{l3}^{\mathrm{T4}} \rightarrow \texttt{l3}^{\mathrm{T1}}$.

The first cycle is not valid because it contains two or more consecutive inter edges. The second cycle is not valid because nodes from thread T1 appear in more than one subsequence. The third cycle is valid and hence indicates a potential deadlock. Specifically, it indicates that the program in Figure 2 can deadlock if thread 1 acquires lock l3 and waits for lock l4 and thread 4 acquires lock l4 and waits for lock l3.

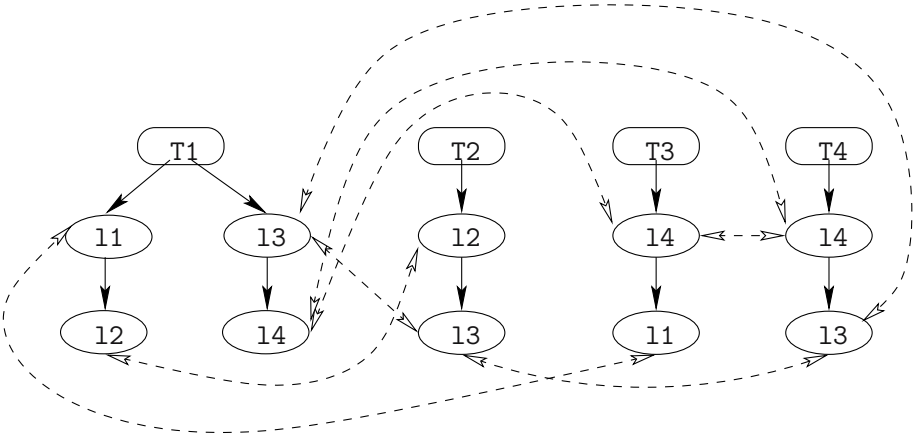

Fig. 1. Run-time lock graph

Now we show that PDL-IGL holds iff the run-time lock graph G contains a valid cycle. Suppose there exist distinct threads $t_0, \ldots, t_{m-1}$ and locks $l_0, \ldots, l_{m-1}$ such that for all $i = 0..m-1$, t_i holds lock l_i while acquiring lock $l_{i+1 \bmod m}$. Let n_i and n'_i denote the nodes in T_i corresponding to the acquire of l_i and the acquire of $l_{i+1 \bmod m}$ nested within it, respectively. Since thread t_i acquires lock l_i and waits for lock $l_{i+1 \bmod m}$, there is a path from n_i to n'_i in run-time lock tree T_i for t_i (because, n'_i is nested below n_i). Note that this path is made of tree edges. The locks l_i and $l_{i+1 \bmod m}$ are distinct, so this path contains at least one tree edge. Also, there

```
Thread 1:
  sync(l1) {
   sync(l2) {
   }
  }
  sync(l3) {
   sync(l4) {
   }
  }
```

```
Thread 2:
  sync(l2) {
   sync(l3) {
   }
  }
```

```
Thread 3:
  sync(l4) {
   sync(l1) {
   }
  }
```

```
Thread 4:
  sync(l4) {
   sync(l3) {
   }
  }
```

Fig. 2. Synchronization behavior of 4 threads. `sync` abbreviates `synchronized`.

is an inter edge from n'_i in run-time lock tree T_i to $n_{i+1 \bmod m}$ in run-time lock tree $T_{i+1 \bmod m}$ in G (by construction). These tree edges and inter edges together form a valid cycle.

Next, we show that existence of a valid cycle C in G implies that the PDL-IGL condition holds. The cycle involves nodes from more than one lock tree, because nodes of a single tree cannot be involved in a cycle. Suppose, C had nodes n_i and n'_i in run-time lock tree T_i for thread t_i, $i \in 0..m-1$ (without loss of generality, we can just consider the beginning and end nodes in the consecutive subsequence from the same thread). Also, nodes n'_i and $n_{i+1 \bmod m}$ are labelled with the same lock (they are consecutive nodes from different lock trees and this is only possible through an inter edge which connects two similar labeled locks). Thus, existence of C implies there exist distinct threads $t_0, \ldots, t_{m-1}$ and locks $l_0, \ldots, l_{m-1}$ (node n_i corresponds to lock l_i and node n'_i corresponds to lock $l_{i+1 \bmod m}$) such that, for all $i = 0..m-1$, t_i holds lock l_i while acquiring lock $l_{i+1 \bmod m}$. Hence, the PDL-IGL condition holds.

Our algorithm to detect existence of a valid cycle traverses all valid paths starting from the root of each lock tree in G using a modified depth-first search (DFS) algorithm, called DFS-ValidCycle, which differs from standard DFS in two ways. First, it traverses only valid paths, because it extends the current path (on the search stack) only with edges satisfying both criteria for validity. Second, a node all of whose neighbors have been explored may be explored multiple times (along incoming inter edges); this is necessary because the set of threads with some lock-tree nodes on the stack might be different on different visits, so the set of valid paths that can be explored by continuing the search from that node is different. The algorithm terminates when a valid cycle is found or all valid paths have been explored. Pseudo-code for the algorithm appears in [AWS05].

To see that the algorithm traverses every valid path, consider a valid path P that begins at a node n in a lock tree T. Extending P by prepending the edges on a path from the root of T to n produces a valid path that is explored by the algorithm when DFS-ValidCycle is started from the root of T. Note that a cycle involving P will be detected, because we check in the algorithm whether n' is anywhere on the stack (not just on the bottom).

To show the worst-case complexity of the algorithm, we consider the number of valid paths in the run-time lock graph. Let $S(k)$ be the number of valid

paths in k lock trees $T_1, \ldots, T_k$, assuming the path visits those lock trees in that order. Then $S(k) = S(k-1) + N_k \times N_{k-1}$, where N_k and N_{k-1} are the number of nodes in lock trees T_k and T_{k-1} respectively, because for each node n in T_{k-1}, the valid paths ending at n can be extended in N_k different ways. Thus, the total number of valid paths is $O(|V|^{|Thread|})$, where $|V|$ is the total number of nodes in the graph, and $|Thread|$ is the total number of threads. There are $|Thread|!$ permutations of $T_1, \ldots, T_k$, and each step of extension or backtracking takes constant time, so the overall worst-case complexity of this algorithm is $O(|V|^{|Thread|} \times |Thread|!)$.

The algorithm can be optimized by observing that many valid paths share a common suffix. Define an ordering on edge types: tree-edge $\geq$ inter-edge. This reflects the fact that in the definition of validity, a tree edge implies fewer restrictions on the next edge in the path. For each node n, n.`visits` is a set of pairs <ts, et>, where ts is a set of threads, and et is an edge type. The meaning of <ts, et> $\in n$.`visits` is that n has been visited along an edge with type et with a stack containing nodes from the lock trees of the threads in ts. If we start the modified DFS at every node n, we do not need to explore a node n' if n'.`visits` contains a pair <$ts1, et$> such that the current stack contains all nodes from the lock trees of the threads in $ts1$ and n' is being visited along an edge with type less than or equal to et. If those conditions hold, then no valid cycles are reachable by continuing the search from n'. This is because there is no valid path from n' back to n that avoids the lock trees on the stack, because if there were, the search would have detected the cycle (containing n and n') and terminated during the visit that added that tuple to n'.`visits`. Pseudo-code for the optimized algorithm appears in [AWS05].

The worst-case complexity of the optimized algorithm is $O(2^{|Thread|} \times |V|^3)$, It is easy to see that each node can have $O(2^{|Thread|})$ items in its visits set. Hence, each node can be explored $O(2^{|Thread|})$ times and during each visit it may need to visit its out-edges. There are at most $|V|$ out-edges from each node. Since we repeat the algorithm for each node, the overall worst-case complexity of the algorithm is $O(2^{|Thread|} \times |V|^3)$.

If the number of threads is a constant, then the algorithm is polynomial in the number of nodes in the run-time lock graph.

However, the algorithm does not consider gate locks and therefore produces false alarms whenever some common lock acquired by at least two threads prevents deadlocks. To eliminate these false alarms, we extend the algorithm to check whether there exist distinct $t_0 \ldots t_{m-1}$ and locks $l_0 \ldots, l_{m-1}$ such that for all $i = 0..m-1$, t_i holds lock l_i while acquiring lock $l_{i+1 \ mod \ m}$ and there do not exist t_i, t_j, and l such that t_i and t_j hold l when acquiring l_i and l_j, respectively. (Such a lock l is called a *gate lock* for the cycle). We call this the Potential for Deadlocks from Locks (PDL) condition.

To check the PDL condition, we modify the algorithm to backtrack (instead of halting) when a valid cycle is encountered, so the algorithm explores all valid cycles, and we check for every valid cycle generated whether there is a gate lock,

i.e., whether no two nodes in different run-time lock trees have ancestors labeled with the same lock. This can be done in $O(|V|^2 \times |Lock|)$ time for each valid cycle , where $|Lock|$ is the number of locks. If a valid cycle without a gate lock is found, potential for deadlock is reported.

3 Deadlock Types

Boyapati, Lee and Rinard [BLR02] introduce a static type system that ensures Java programs are deadlock-free. The deadlock types express a partial order among the locks. The typing rules ensure that whenever a thread holds multiple locks, the thread acquires the locks in descending order. This ensures absence of cyclic waiting and therefore implies absence of deadlocks.

The rest of this section briefly describes Parameterized Race Free Java (PRFJ) [BR01], and [BLR02]'s deadlock types. In PRFJ, as in its predecessor Race Free Java [FF00], types indicate the synchronization discipline (also called "protection mechanism" or "owner") used to co-ordinate accesses to each object. To allow different instances of a class to use different protection mechanisms, each class is parameterized by formal owner parameters, which may be instantiated with other formal owner parameters, final expressions (*i.e.*, expressions whose value does not change) representing locks, or special owners (described below).

A final expression used as an owner specifies a lock that must be held when the object is accessed. There are four special owners: `thisThread`, `self`, `readonly` and `unique`. `readonly` indicates that the object is readonly and cannot be updated. `unique` means that there is a unique reference to the object. `thisThread` means that the object is thread-local (*i.e.*, unshared). `self` means that the object is protected by its own lock. The owner of an object is said to *guard* all of its fields.

Method declarations may have a `accesses` clause that contains a set of final expressions; the owners of these expressions are locks, those locks must be held when the method is invoked.[1]

Deadlock types associate a lock level with each lock. The typing rules ensure that if a thread acquires a lock l_2 (which the thread does not already hold) while holding a lock l_1, then l_2's level is less than l_1's level; in other words, locks are acquired in descending order. Lock levels and the partial order on them are defined by statements of the form `LockLevel` $\mathtt{l}_1$ `= new;` $\mathtt{l}_2$ `<` $\mathtt{l}_1$. In PRFJ, only locks on objects with owner `self` can be acquired (acquiring locks on other objects is not useful for showing race-freedom), so lock levels are associated only with objects with owner `self`.

In this paper, we focus on [BLR02]'s basic deadlock types, in which all instances of a class are associated with the same lock level. An extension that supports polymorphism in lock levels, i.e., that allows classes to be parameterized with formal lock level parameters, is presented in [BLR02], but in our experience, this extra flexibility is rarely useful, and it makes type inference much more difficult.

[1] For simplicity, we ignore the distinction between owners and root owners in this overview.

```
class Account<self:l1> {
  int balance;
  Vector<self:l2> v = new Vector<self:l2>();
  Locklevel l2 < l1;

  int deposit(int x, int tid) locks l1, l2 {
    synchronized(this) {
      this.balance = this.balance + x;
      v.addElement(new Integer<readonly>(tid));
    }
  }
}

class Vector<self:l2> {

....

  synchronized void ensureCapacity(int minCapacity) locks l2, this {...}
  synchronized void addElement(Object<f> obj) locks l2 {
    if (elementCount == elementData.length)
      ensureCapacity(elementCount + 1);
    modCount++;
    elementData[elementCount++] = obj;
  }
}

Account<self:l1> a1 = new Account<self:l1>;
fork(a1){a1.deposit(100,1);}
fork(a1) {a1.deposit(100,2);}
```

Fig. 3. An example program with race-free types and deadlock types

In the deadlock type system, each method m is annotated with a locks clause that contains a set of lock levels. These lock levels are the maxima amongst the levels of locks that may be acquired when m is executed. To ensure that a program is free of deadlocks, the typing rule for method calls ensures that the caller only holds locks that are of a higher level than the levels in the called method's `locks` clause. A `locks` clause may also contain a lock l, which indicates that the thread invoking the method may hold a lock on object l. The typing rule for synchronized expression checks that the lock being acquired is l or has a lower level than l. This allows typing of programs in which, for example, a synchronized method of a class calls a synchronized method of the same class on the same object.

The program in Figure 3 illustrates race-free types and deadlock types. It shows a class `Account` whose owner is `self` with lock level `l1`. It has an instance field `v` of class type `Vector` with owner `self` and lock level `l2`. The main thread spawns two threads, each of which invoke the `deposit` method on account `a1`. The `deposit` method acquires a lock on `this` followed by a lock on its field

v when the addElement method of v is invoked. This is consistent with the declared lock level ordering l2 < l1, since the lock on v is acquired after the lock on a1. The lock levels specified in the locks clause of deposit method satisfy the method invoke rule as it is called with no locks held (corresponding to lock level infinity which is greater than both l1 and l2). The addElement method of v is called with current lock level (the lock level of the most recently acquired lock but not yet released) l1 which is greater than l2 specified in the locks clause. Hence, the call to addElement also typechecks. When the addElement method calls the ensureCapacity method of v, the current lock level l2 is not greater than l2, rather it is the same. However, the ensureCapacity method also contains a lock, viz., this (which has lock level l2), in the locks clause which is held at the call site. The program typechecks because the typing rule for method calls allows the current lock level to be the same as the level of the lock l in the locks clause if the thread invoking the method already holds the lock on l. Reacquiring the lock on this in ensureCapacity method also typechecks, because the typing rule for synchronized expressions checks whether the newly acquired lock is the same as specified in the locks clause.

4 Static Type Inference for Deadlock Types

The following section presents a type inference algorithm for [BLR02]'s basic deadlock type system. The algorithm assumes that race-free types are already known. Type inference for race-free types is NP-hard, but in practice, race-free types can often be obtained using a SAT solver [FF04] or type discovery [AS04, ASS04].

The algorithm works as follows:

1. Each field, method parameter and local variable with owner self is initially assigned a distinct lock level. The levels are initially unordered. For each method, equality constraints among lock levels are generated based on assignment statements and method invocations. This is necessary for programs to typecheck as left and right hand side of an assignment must have the same type (modulo subtyping), and the type now includes the lock level when the owner is self. Similarly, for each call site, each argument to the called method must have the same lock level as the corresponding parameter. The constraints can then be solved using the standard Union-Find algorithm.
2. A static lock graph G_L is constructed that captures the locking pattern of the program. A synchronized statement is redundant if the final expression corresponding to the lock acquire appears nested below a lock acquire of the same final expression by a synchronized statement in the same method or if the final expression is the same as the rootowner of an expression in the accesses clause of the method containing this synchronized statement. For each synchronized statement in the program that is not redundant (including the implicit synchronized statement enclosing the body of each synchronized method), the graph contains a corresponding node, called a

lock node . For each method m in the program, the graph contains a corresponding node n_m, called a *method node*. There is an edge from a lock node n_1 to a lock node n_2 if the synchronized block corresponding to node n_2 is syntactically directly nested within the synchronized block corresponding to n_1 or the other synchronized statements between n_1 and n_2 are redundant. There is an edge from each method node n_m to the lock node for each outermost `synchronized` block in m. For each method call within the scope of a synchronized block except calls to `Thread.start`, there is an edge from the lock node corresponding to the inner most synchronized block that encloses the method call to the method node for the called method.

3. The method node n_m for method m is associated with a set L_m of lock levels. L_m^{init} contains the lock levels of the lock nodes that are children of n_m. Recall that each lock node is associated with a unique lock level in step 1. If n_m has no lock node children, L_m^{init} is empty. Let $called(m)$ denote the set of methods directly called by method m such that the corresponding call sites are not in the scope of a synchronized block in m. The set L_m for each method is computed using a fixed point computation. It is the least fixed point solution to the following set of constraints: for each method m, $L_m = L_m^{init} \cup \bigcup_{m' \in called(m)} L_{m'}$. The right side is monotonic in $L_{m'}$, so the least solution can be computed by a standard fixed-point computation. For each method node n_m, the lock levels in L_m are added to the `locks` clause of m.
4. For each lock node n with lock level l, and for each lock node n' with lock level l', such that there is an edge from n to n', add the declaration $l > l'$ to the inferred typing. If there is an edge from n to a method node n_m, then for each lock level l' in L_m other than l, add the declaration $l > l'$ to the inferred typing.
5. For each lock node n with lock level l in method m, if n is reachable from a lock node ancestor of n with the same lock level and in a different method, then add to the `locks` clause of m the final expression denoting the lock acquired by the `synchronized` statement corresponding to n. (Leaving lock level l in m's `locks` clause is sometimes unnecessary but harmless).

The complexity of the algorithm is $O((|V_m|^3) + ((|V_m| + |V_l|) \times |E|))$, where $|V_m|$ is the number of method nodes (equal to the number of methods in the program), $|V_l|$ is the number of lock nodes (equal to number of synchronized statements in the program), and $|E|$ is the number of edges in G_L. Our type inference algorithm is correct in the sense that it produces correct typings for all typable programs, as shown in the appendix. For untypable programs, our inference algorithm does not simply halt and report failure; rather, it produces the best type annotations it can for the given program. This is useful for focused run-time checking, as described in Section 6. Our type inference algorithm does not attempt to determine whether the given program is typable; instead, we simply run the type checker on the program with the inferred type annotations.

Figure 4 shows a static lock graph G_L for program in Figure 3. It contains method nodes for `deposit`, `addElement` and `ensureCapacity`. It contains a

lock node for each `synchronized` statement. Each lock node is labeled with the acquired lock and its lock level. For example, n_1 corresponds to the `synchronized` statement in method `deposit`, which acquires the lock on `this`, which has lock level `l1` (from step 1 of the type inference algorithm). Each method node n_m is labeled with the set $L(m)$ computed in step 3 of the type inference algorithm.

After computing G_L and L, the type inference algorithm infers the deadlock types for the program in Figure 3 as follows. First the elements of $L(m)$ are used in the `locks` clause of method m. For example, the `locks` clause of method `deposit` contains lock level `l1`, since $L(\texttt{deposit}) = \{\texttt{l1}\}$. Edges from lock nodes to method nodes introduce lock level orderings. For example, the edge from n_1 to $n_{\texttt{addElement}}$ introduces the declaration `l1 > l2` by step 4. n_2 is an ancestor of n_3 with the same lock level m and in a different method. Therefore, step 5 adds `this` to the `locks` clause for method `ensureCapacity`.

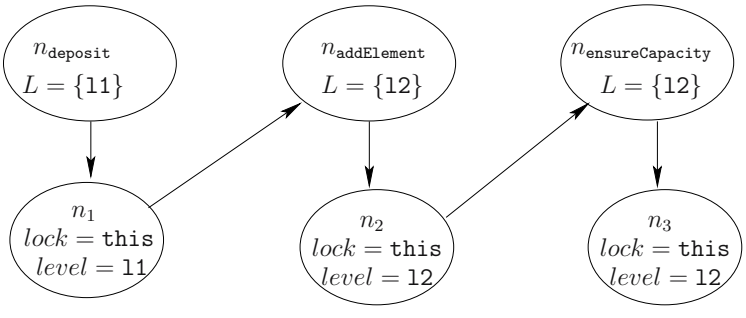

Fig. 4. Static lock graph G_L for program in Figure 3

Theorem 1. *The algorithm above produces a correct typing for a program if the program is typable in the basic deadlock type system of [BLR02].*

The proof for the theorem above is available in the companion technical report [AWS05].

5 Deadlock-Free Extended Parameterized Atomic Java

This section describes how to add basic deadlock types [BLR02], discussed in Section 3, to Extended Parameterized Atomic Java (EPAJ) [SAWS05], our type system that ensures absence of races and atomicity violations. The resulting type system, Deadlock-Free Extended Parameterized Atomic Java (DEPAJ), ensures absence of deadlocks due to locks and hence provides stronger atomicity guarantees. Atomicity types in EPAJ are adopted from Atomic Java [FQ03] and are based on Lipton's theory of reduction [Lip75], which requires that the code to be reduced (*i.e.*, shown to be atomic) cannot be involved in a deadlock. EPAJ and Atomic Java do not consider whether the code can be involved in deadlocks. By adding deadlock types [BLR02] to EPAJ, the resulting type system provides

stronger atomicity guarantees. Adding the deadlock types proposed in [BLR02] to EPAJ is straightforward. For brevity, we do not describe atomicity types here, since they are not changed by the addition of deadlock types, although they get a stronger semantics.

EPAJ extends PRFJ to allow each field to have a different guard. Because ownership in EPAJ is per field, not per object (as in PRFJ), PRFJ's notion of rootowner is not well-defined, so we discard it and compensate by allowing formal owner parameters in `accesses` clauses, which are called `requires` clauses in RFJ [FF00] and EPAJ. To make this work, every formal owner parameter f is qualified with a final expression e that indicates the object that f refers to when f is instantiated with `self`. A `guarded_by` clause on a field contains a *lock expression*, which is either a final expression or $f\$e$ where e is a final expression and f is the first formal owner parameter in the type of e. A `guarded_by` clause cannot contain a special owner (explained in Section 3) explicitly, but the formal owner parameter in it may be instantiated with a special owner, providing the same effect. PRFJ allows synchronization only on objects with owner `self`, because only those objects can be roots of ownership trees. In contrast, EPAJ eliminates the concept of root owner and consequently allows synchronization on objects with any owner. Therefore, in DEPAJ, a lock level is associated with each final expression used in a `synchronized` statement. This has the side-effect of allowing different lock levels for different instances of a class in some cases, even in the basic (non-polymorphic) deadlock type system.

The type inference algorithm in Section 4 easily carries over to infer deadlock types in DEPAJ.

6 Focusing Run-Time Checks for Deadlock Detection

Deadlock types enforce a conservative strategy for preventing deadlocks. Therefore, there are deadlock-free programs not typable in this type system. For example, programs which have cycles in the static lock level ordering are untypable even though they may be deadlock-free. An example appears below. The type system also requires all elements of a Collection class to have the same lock level. This may be too restrictive and can lead to untypable programs even though the programs are deadlock-free. For such programs, information gathered from the type system can be used to focus run-time checking, *i.e.*, run-time checking can safely be omitted for parts of the program guaranteed to be deadlock-free by the type system.

To focus the generalized version of the GoodLock algorithm that does not handle gate locks, we find all the cycles of the form $l_1 > l_2 ... > l_1$ among lock level orderings produced by the deadlock type inference algorithm. We instrument only lock acquires and releases of expressions whose lock level is part of a cycle. Other synchronized expressions do not need to be instrumented. This leads to fewer intercepted events and smaller lock trees that need to be analyzed. It is easy to determine which lock levels are part of cycles. Construct a graph $G = (V, E)$, where each lock level is a node in V and there is an edge from l to l' if the

inferred typing declares $l > l'$. A simple depth first search can find all nodes that are part of some cycle.

As an example, we consider a modified version of the elevator program, developed at ETH Zürich and used as a benchmark in [vPG01]. elevator is a simple discrete-event simulation of people going up and down in elevators; we extended it to model the people explicitly as objects. The instances of `Person` are initially stored in a static `Vector` field `people` in the main `Elevator` class. When some of them make a request to go up or down they are moved from `Elevator.people` to the `upPeople` or `downPeople` vector of the appropriate instance of `Floor`. An instance of `Lift` services the request by acquiring the lock on the instance of `Floor` where the requester(s) are waiting, updating the status flags of the floor, and then moving people from the `upPeople` or `downPeople` field of the floor to the appropriate `peopleFor` vector in the `Lift` instance based on their destination floor. On reaching the destination floor, the lift moves the people in the corresponding `peopleFor` vector back to the `Elevator.people` vector. All the moves between vectors are done using the `addAll` method of the `Vector` class. `v1.addAll(v2)` adds each object in `v2` to `v1`. `v1.addAll(v2)` acquires the lock on `v2` while holding the lock on `v1`. The modified elevator program is deadlock-free, but not typable with [BLR02]'s basic deadlock types. This is because every `Vector` class is self-synchronized with some lock level, say l. However, the lock level orderings required by the typing rules as a result of the calls to `addAll` together create a cycle in the lock ordering. The program is not typable even in the full polymorphic version of the type system, for essentially the same reason. If the vectors in `Elevator.people`, `Lift.peopleFor`, and `Floor.upPeople` are assigned different lock levels l_1, l_2 and l_3 in the polymorphic type system, then the orderings $l_1 > l_2 > l_3 > l_1$ are required. This cycle makes the program untypable. Different instances of `Lift` are started in a loop, so it is not possible even in the polymorphic type system to assign different lock levels to the `peopleFor` field of different instances of `Lift`.

Our type inference algorithm infers orderings of the form $l > l$, where l is the lock level assigned to the self synchronized `Vector` class. Other locks, including locks on instances of `Floor` and `Lift`, are not involved in the cycle. So, we focus run-time checks by intercepting lock acquires and releases only on instances of `Vector`.

To focus the generalized version of the GoodLock algorithm that handles gate locks, we find all the cycles among lock level orderings produced by the type inference algorithm as discussed above. All lock levels that are comparable to lock levels involved in a cycle in the ordering of lock levels need to be instrumented (not just the lock levels involved in a cycle).

7 Experience

To evaluate our technique, we manually ran the inference algorithm on the five multi-threaded server programs used in [BR01]. The programs are small, ranging from about 100 to 600 lines of code, and totaling approximately 1600 LOC.

Our type inference algorithm successfully inferred complete and correct typings for all five server programs. We compared the inferred typings to Boyapati's manually inserted type annotations. In his code, ChatServer contains 12 deadlock annotations, GameServer contains 8 deadlock annotations, HTTPServer contains 2 deadlock annotations, and QuoteServer and PhoneServer contain none. Thus, approximately 1600 LOC required 22 deadlock annotations; that's approximately 15 annots/KLOC. Our type inference algorithm eliminates the need for the user to write all of those annotations.

Since these programs are guaranteed deadlock free using the types, the need for run-time checking for potential deadlocks is completely eliminated for these programs.

Table 1. The run times of dynamic deadlock detection for modified elevator example

		3 threads	7 threads	15 threads	30 threads	60 threads
Base time		0.23s	0.30s	0.52s	2.60s	6.60s
Full	Size	621	1037	1848	3359	5734
	Unopt	0.76s	0.94s	14.93s	1m23.08s	3m42.9s
	Opt	1.10s	1.66s	17.32s	1m28.05s	4m3.0s
Focused	Size	433	646	1063	1824	2947
	Unopt	0.40s	0.53s	11.71s	34.91s	1m22.66s
	Opt	0.51s	0.72s	12.40s	36.35s	1m28.28s

We implemented the unoptimized and optimized generalized Goodlock algorithms without gate locks described in Section 2 and used them to analyze the modified `elevator` program described in Section 6. The experiments were performed on a Sun Blade 1500 with a 1GHz UltraSPARC III CPU, 2GB RAM, and JDK 5.0. Table 1 shows the running times for the `elevator` program with 3,7,15, 30 and 60 `Lift` threads. The "Base time" row gives the execution time of the original program without any instrumentation. The "Full" and "Focused" rows give the execution results of the program augmented with full and focused run-time checking, respectively. For "Full" and "Focused" rows, sub rows "Size", "Unopt" and "Opt" give the the number of nodes in all runtime lock trees, execution times of the unoptimized algorithm, and optimized algorithm respectively. As discussed in Section 6, focused run-time checking in this example intercepts lock acquires and releases only on instances of `Vector`. The results demonstrate that the focused analysis significantly decreases the run-time overhead of deadlock checking and the size of runtime lock trees. Let Ofull = Full − Base denote the overhead of full checking, and Ofoc = Focused − Base denote the overhead of focused checking. The average speedup (i.e., fractional reduction in overhead) is (Ofull − Ofoc)/Ofull, which is 55.8% for the unoptimized algorithm, and 58.4% for the optimized algorithm. The average size of the runtime lock trees is reduced by 41%. Surprisingly, the optimized algorithm runs slower than the unoptimized algorithm, although its asymptotic worst-case time complexity is better. The main reason is that the optimized algorithm uses more complicated data structures, and for `elevator` example, where the the run-time lock graph is relatively

simple, the benefit of caching explored paths falls short of the overhead of data structure maintenance.

8 Related Work

Techniques for run-time detection of deadlocks include the GoodLock algorithm [Hav00], a run-time analysis implemented in Compaq's Visual Threads [Har00] and ConTest [EFG+03]. As discussed in Section 2, the GoodLock algorithm detects only potential deadlocks involving two threads. We generalize it to detect deadlocks involving any number of threads. Bensalem and Havelund independently generalized the GoodLock algorithm to detect such deadlocks [BH05]. They do not consider combining it with static analysis; we do. However, their algorithm eliminates false alarms arising from cycles that contain lock acquires and releases that cannot happen in parallel. Visual Threads can detect potential for deadlocks [Har00], but it is not clear what algorithm is used. ConTest [EFG+03] detects actual deadlocks, not potential deadlocks, and therefore may miss some potential deadlocks; on the other hand, ConTest's scheduling perturbation heuristics make potential deadlocks of all kinds (including deadlocks due to condition synchronization) more likely to manifest themselves as actual deadlocks during testing with ConTest, compared to testing without ConTest. A recent extension to ConTest implements a run-time deadlock checking algorithm that combines information obtained from multiple executions of the program [FNBU]. That technique is compatible with our work, and it would be useful to combine them. ConTest does not use static analysis to optimize run-time checking.

Hatcliff *et al.* [HRD04] verify atomicity specifications using model checking. Their approach is more accurate than type-based analysis of atomicity: it does not produce false alarms, and it fully enforces the condition that deadlock must not occur within an atomic block. Hatcliff et al. point out that previous type systems for atomicity do not enforce this condition. DEPAJ takes a step towards addressing this, by ensuring that lock-induced deadlocks do not occur within atomic blocks. Since their approach is based on model checking, it is computationally expensive and hence practical only for relatively small programs.

Engler *et al.* [EA03], von Praun [vP04], and Williams *et al.* [WTE05] developed inter-procedural static analyses that detect potential deadlocks in programs. These static analyses are also based on checking whether locks are acquired in a consistent order by all threads. These static analyses are more sophisticated and more accurate than basic deadlock types but still produce numerous false alarms (Engler *et al.* and Williams *et al.* partially address this problem by using heuristics to rank or suppress warnings that seem more likely to be false alarms), so it would be useful to use them in conjunction with run-time checking, which generally produces fewer false alarms. Specifically, although these papers do not consider run-time checking, the results of their analyses could be used instead of deadlock types in our technique for focused run-time detection of potential deadlocks.

The idea of using static analysis to optimize run-time checking has been applied to detection of races [vPG01, CLL+02, ASWS05] and atomicity violations [SAWS05, ASWS05] but not to detection of potential deadlocks, to the best of our knowledge.

References

[AS04] Rahul Agarwal and Scott D. Stoller. Type inference for parameterized race-free Java. In *Proceedings of the Fifth International Conference on Verification, Model Checking and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 149–160. Springer-Verlag, January 2004.

[ASS04] Rahul Agarwal, Amit Sasturkar, and Scott D. Stoller. Type discovery for parameterized race-free Java. Technical Report DAR-04-16, Computer Science Department, SUNY at Stony Brook, September 2004.

[ASWS05] Rahul Agarwal, Amit Sasturkar, Liqiang Wang, and Scott D. Stoller. Optimized run-time race detection and atomicity checking using partial discovered types. In *Proc. 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM Press, November 2005.

[AWS05] Rahul Agarwal, Liqiang Wang, and Scott D. Stoller. Detecting potential deadlocks with static analysis and runtime monitoring. Technical Report DAR-05-25, Computer Science Department, SUNY at Stony Brook, September 2005. Available at http://www.cs.sunysb.edu/~ragarwal/deadlock/.

[BH05] Saddek Bensalem and Klaus Havelund. Scalable deadlock analysis of multi-threaded programs. In *Proceedings of the Parallel and Distributed Systems: Testing and Debugging (PADTAD) Track of the 2005 IBM Verification Conference*. Springer-Verlag, November 2005.

[BLR02] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proc. 17th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 211–230, November 2002.

[BR01] Chandrasekar Boyapati and Martin C. Rinard. A parameterized type system for race-free Java programs. In *Proc. 16th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, volume 36(11) of *SIGPLAN Notices*, pages 56–69. ACM Press, 2001.

[CLL+02] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 258–269. ACM Press, 2002.

[EA03] Dawson R. Engler and Ken Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Proc. 24th ACM Symposium on Operating System Principles*, pages 237–252. ACM Press, October 2003.

[EFG+03] Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Framework for testing multi-threaded Java programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003.

[FF00] Cormac Flanagan and Stephen Freund. Type-based race detection for Java. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 219–232. ACM Press, 2000.

[FF04] Cormac Flanagan and Stephen Freund. Type inference against races. In *Proc. 11th International Static Analysis Symposium (SAS)*, volume 3148 of *Lecture Notes in Computer Science*. Springer-Verlag, August 2004.

[FNBU] Eitan Farchi, Yarden Nir-Buchbinder, and Shmuel Ur. Cross-run lock discipline checker for java. Tool proposal for IBM Verification Conference 2005.

[FQ03] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 338–349. ACM Press, 2003.

[Har00] Jerry J. Harrow. Runtime checking of multithreaded applications with Visual Threads. In *Proc. 7th Int'l. SPIN Workshop on Model Checking of Software*, volume 1885 of *Lecture Notes in Computer Science*, pages 331–342. Springer-Verlag, August 2000.

[Hav00] Klaus Havelund. Using runtime analysis to guide model checking of java programs. In *Proc. 7th Int'l. SPIN Workshop on Model Checking of Software*, volume 1885 of *Lecture Notes in Computer Science*, pages 245–264. Springer-Verlag, August 2000.

[HRD04] John Hatcliff, Robby, and Matthew B. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model checking. In *Proceedings of the Fifth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2004)*, Lecture Notes in Computer Science. Springer-Verlag, 2004.

[Lip75] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.

[SAWS05] Amit Sasturkar, Rahul Agarwal, Liqiang Wang, and Scott D. Stoller. Automated type-based analysis of data races and atomicity. In *Proc. ACM SIGPLAN 2005 Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM Press, June 2005.

[vP04] Christoph von Praun. *Detecting Synchronization Defects in Multi-Threaded Object-Oriented Programs*. PhD thesis, ETH Zürich, 2004.

[vPG01] Christoph von Praun and Thomas R. Gross. Object race detection. In *Proc. 16th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, volume 36(11) of *SIGPLAN Notices*, pages 70–82. ACM Press, October 2001.

[WTE05] Amy Williams, William Thies, and Michael D. Ernst. Static deadlock detection for Java libraries. In *Proc. 2005 European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science. Springer-Verlag, July 2005.