**Noname manuscript No.**
(will be inserted by the editor)

# Symbolic Modular Deadlock Analysis

**Jyotirmoy V. Deshmukh** · **E. Allen Emerson** ·
**Sriram Sankaranarayanan**

Received: Jun 14 2010 / Accepted:

**Abstract** Methods in object-oriented concurrent libraries often encapsulate internal synchronization details. As a result of information hiding, clients calling the library methods may cause thread safety violations by invoking methods in an unsafe manner. This is frequently a cause of deadlocks. Given a concurrent library, we present a technique for inferring *interface contracts* that specify permissible concurrent method calls and patterns of aliasing among method arguments. In this work, we focus on deriving contracts that guarantee deadlock-free execution for the methods in the library. The contracts also help client developers by documenting required assumptions about the library methods. Alternatively, the contracts can be statically enforced in the client code to detect potential deadlocks in the client. Our technique combines static analysis with a symbolic encoding scheme for tracking lock dependencies, allowing us to synthesize contracts using an SMT solver. Additionally, we investigate extensions of our technique to reason about deadlocks in libraries that employ signaling primitives such as *wait-notify* for cooperative synchronization. Our prototype tool analyzes over a million lines of code for some widely-used Java libraries within an hour, thus demonstrating its scalability and efficiency. Furthermore, the contracts inferred by our approach have been able to pinpoint real deadlocks in clients, *i.e.* deadlocks that have been a part of bug-reports filed by users and developers of client code.

**Keywords** Deadlock Prediction · Static Analysis · Concurrent Libraries

Part of this paper was published in the proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering, 2009, [10].

J. V. Deshmukh (✉)
Department of Electrical Engineering, The University of Texas at Austin, Austin, TX 78712
E-mail: jyotirmoy@cerc.utexas.edu

E. A. Emerson
Department of Computer Science, The University of Texas at Austin, Austin, TX 78712
E-mail: emerson@cs.utexas.edu

S. Sankaranarayanan
Department of Computer Science, University of Colorado Boulder, Boulder, CO 80309-0430
E-mail: srirams@colorado.edu

## 1 Introduction

Concurrent programs are prone to a variety of thread-safety violations arising from the presence of *data races* and *deadlocks*. In practice, as data races are abundant and difficult to debug, they have garnered considerable attention from the program analysis community. A *knee-jerk* response to avoiding race conditions is evident in the prolific use of locking constructs in concurrent programs. Languages such as `Java` have promoted this by providing a convenient `synchronized` construct to specify mutual exclusion with monitors. Locking is sometimes naively used as a "safe" practice, rather than as a requirement. Overzealous locking not only causes unnecessary overhead, but can also lead to unforeseen deadlocks. Deadlocks can severely impair real-time applications such as web-servers, database systems, mail-servers, device drivers, and mission-critical systems with embedded devices, and typically culminate in loss of data, unresponsiveness, or other safety and liveness violations.

In this paper, we focus on deadlocks arising from circular dependencies in synchronization constructs such as locks and signaling primitives. Languages such as `Java` combine the mutual exclusion provided by locks with the cooperative synchronization provided by signaling primitives into a single *monitor* construct. In this paper, we use the abstract term lock to mean both specialized lock variables in languages such as `C`, `C++/pthread`, and monitors used for enforcing mutual exclusion in `Java`.

Deadlock detection is a well-studied problem, and both static and dynamic approaches have been proposed [1, 5, 8, 14, 24, 26, 30]. Typically, such techniques construct *lock-order graphs* that track dependencies between locks for each thread. Lock-order graphs for concurrent threads are then merged, and a cycle in the resulting graph indicates a possibility of a deadlock. Such techniques typically assume a *closed system*, and are thus useful for detecting *existing deadlocks* in a given application.

However, most software is designed compositionally and treating individual components as closed systems could lead to potential deadlocks being undetected. In particular, consider the now prevalent *concurrent libraries*, *i.e.*, collections of modules that support concurrent access by multiple client threads. Modular design principles mandate that the onus of ensuring thread safety rests with the developer of such a library. This has an undesirable side-effect: several details of synchronization are obscured from the developer of client code that makes use of this library. Consequently, the client developer may unintentionally invoke library methods in ways that can cause deadlocks.

Analyzing concurrent libraries for deadlocks has two main aspects: First of all, we wish to identify if, for *any* client, there are library methods that can be concurrently called in a manner that causes a deadlock. This is termed the *deadlockability* problem. Secondly, we wish to use the results of this analysis to search for the existence of deadlocks in a *particular* client that invokes these library methods. Deadlockability analysis was first introduced by Williams et al. [30]. Therein, the authors construct a lock-order graph for each library method. The *types* of syntactic expressions corresponding to object monitors are used as conservative approximations for the may-alias information between these monitors. The authors show that their approach helps in identifying important potential deadlocks; however, their approach is susceptible to false positives, which have to be then filtered using (possibly unsound) heuristics.

This paper addresses the same underlying problem as that of Williams et al. [30], under similar assumptions about the underlying language (`Java`), concurrent libraries, their clients, and the use of synchronization. The key contributions of this paper are as follows:

```
public class EventQueue {
   EventQueue nextQueue;
   void postEventPrivate (Event e) {
      . . .
1:    synchronized (this) {
2:       nextQueue.postEventPrivate(e);
      }
      . . .
   }
   void push (EventQueue eq) {
      . . .
3:    nextQueue = eq;
      . . .
   }
   void wakeup(boolean f) {
      . . .
4:    synchronized (this) {
5:       nextQueue.wakeup(f);
      }
      . . .
   }
```

Fig. 1.1: Methods in `java.awt.EventQueue`

(a) We reason about possible aliasing patterns between nodes in a lock graph explicitly rather than with type-based approximations.
(b) We reduce the space of possible aliasing patterns between nodes by using a notion of subsumption between aliasing patterns. This enables a symbolic approach for enumerating aliases between nodes using SAT-modulo Theory (SMT) solvers. The focus on aliasing patterns allows us to rule out infeasible aliases by means of a prior pointer analysis. Overall, our analysis is just as scalable while producing fewer false positives.
(c) We synthesize logical expressions involving aliasing between the parameters of concurrent method invocations such that these expressions guarantee deadlock-free execution of the library methods. These contracts can then be used to detect deadlocks in a particular client.
(d) We identify usage patterns of *wait-notify* based synchronization that can lead to potential deadlocks.

## 1.1 Approach at a Glance

To illustrate the problem with treating libraries as closed systems, we use the `Java` code snippet (shown in Fig. 1.1) from the `EventQueue` class in `Java`'s `awt` library. In Lines 1 and 4, the "`synchronized(this)`" statement has the effect of acquiring a lock on the "`this`" object. The `nextQueue` variable is a data member of the `EventQueue` class, which is set by the `push` method (Line 3). By design, the `postEventPrivate` and `wakeup` methods are intended to perform their action on the `EventQueue` instance "`this`", on which they are invoked, and then act on "`this.nextQueue`" (Lines 2 and 5). Consider the case wherein one client thread (say $T_1$) invokes "`a.push(b)`", while another client thread (say $T_2$) invokes "`b.push(a)`". Subsequently, if $T_1$ invokes "`a.postEventPrivate(e)`" concurrently while $T_2$ simultaneously

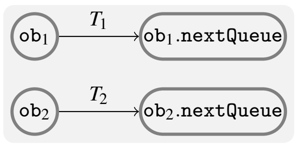
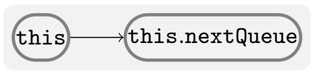


Fig. 1.2: Joint lock-order graph from the `postEventPrivate` and `wakeup` methods.

invokes "`b.wakeup(true)`", then this may result in a deadlock. This deadlock can manifest itself in real client code, as reported by client developers in [9] (bug-id: 6542185).

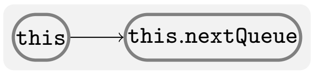
Our deadlockability analysis first performs static inspection of the given concurrent library to identify lock-order graphs for each method. The lock-order graph for the `wakeup` method in Fig. 1.1 captures the acquisition of the lock for the "`this`" object followed by that of the "`this.nextQueue`" object:

`this.wakeup(..)`:



Similarly, `postEventPrivate` method first acquires a lock on the "`this`" object followed by the "`this.nextQueue`" object, yielding an identical acyclic lock-order graph:

`this.postEventPrivate(..)`:



Consider a client that performs concurrent calls to the methods from two different threads on objects: $\texttt{ob}_1, \texttt{ob}_2$:

$$T_1 : \texttt{ob}_1\texttt{.wakeup(true)} \parallel T_2 : \texttt{ob}_2\texttt{.postEventPrivate()}$$

Assuming no other lock acquisitions are made by the threads themselves, no other calls to methods and no aliasing/sharing between the objects, the lock-order graph of the client is as shown in Fig. 1.2.

Normally, the two graphs by themselves are acyclic, and the method calls by themselves do not seem to cause an obvious deadlock. However, the lock-order graph above assumes that the objects $\texttt{ob}_{1,2}$ are not aliased/do not share fields. Consider, on the other hand, the scenario wherein the object $\texttt{ob}_1\texttt{.nextQueue}$ aliases $\texttt{ob}_2$ and $\texttt{ob}_2\texttt{.nextQueue}$ aliases $\texttt{ob}_1$. Under such a scenario, the lock-order graph of Fig. 1.2 is modified by fusing the aliased nodes into a single node to obtain the graph depicted in Fig. 1.3. This graph clearly indicates the possibility of a deadlock. Furthermore, prior calls to the `push` methods set up the required pattern of aliasing along the lines of [9] (bug-id:6542185). It is important to note that techniques that assume a closed system would only generate the lock-order graph shown in Fig. 1.2, and would thus miss a potential deadlock.

At a broad level, the techniques developed in this paper provide a practical framework to:

(a) identify potential deadlock situations by *efficiently* considering all feasible aliasing and sharing scenarios between objects at the concurrent call-sites of library methods,

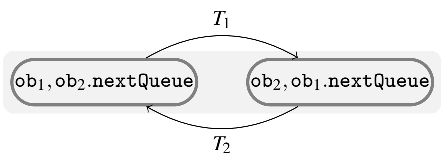


Fig. 1.3: Lock-order graph for $T_1||T_2$ under aliasing of nodes.

(b) derive an *interface contract* that characterizes safe aliasing patterns for concurrent calls to library methods.

Concretely, our technique synthesizes the aliasing scenario described above. For calls to the `wakeup` and the `postEventPrivate` methods, our analysis derives the contract specifying that at any concurrent call to a.`wakeup` () and b.`postEventPrivate` (), the aliasing between `a`,`b` must satisfy: $\neg$isAliased(`a,b.nextQueue`) $\vee\neg$isAliased(`b,a.nextQueue`)

This is sufficient to guarantee deadlock-free execution of these methods assuming that the synchronization operations of the client cannot "interfere" with that of the library.

The layout of the paper is as follows: In Sec. 2, we introduce the problem of deadlock detection for concurrent libraries and discuss the notation. In Sec. 3, we introduce a symbolic encoding scheme for representing lock acquisition orders in library methods, given an aliasing pattern spanning the objects relevant to the methods. In Sec. 4 we show how we can identify all potential deadlocks for a library by optimally enumerating all aliasing patterns. We discuss usage patterns for *wait-notify* monitors that can potentially lead to deadlocks in Sec. 5. We present an encoding for such patterns that allows us to similarly perform symbolic reasoning about such deadlocks in Sec. 6. In Sec. 7, we show how interface contracts derived for a given library can be used compositionally while analyzing the client of the library for deadlocks. Experimental results obtained by analyzing well-known `Java`-based libraries are discussed in Sec. 8. Finally, we discuss related work, and conclude in Sec. 9.

## 2 Preliminaries

We assume that we are given a concurrent library written in a *class-based* object-oriented programming language such as `C++` or `Java`. In the following discussion, we introduce the type-based semantics and the concurrency model for such libraries, loosely adhering to the model used in `Java`.

*Library and Types.*
Formally, we define a library $\mathscr{L}$ as a collection of *class* definitions $\langle\mathscr{C}_1,\ldots\mathscr{C}_k\rangle$. Each class $\mathscr{C}_i$ denotes a corresponding *reference type* $C_i$. A class definition consists of definitions for *data members* (also called fields), and *methods* (member functions). We say that $C_2$ is a *subtype* of $C_1$ if $\mathscr{C}_2$ is a subclass of $\mathscr{C}_1$.

Data members have *primitive types* (`int`, `double`, *etc.*), or reference types[1]. An object is an instance of a class $\mathscr{C}_i$, and the type of the object is the corresponding reference type $C_i$. Let $V = \{\texttt{ob}_1,\ldots,\texttt{ob}_k\}$ be a (super-)set of all the object variables (*references* in `Java` terminology) occurring in the methods of interest in $\mathscr{L}$.

[1] Apart from class types, array types are also classified as reference types, and our technique handles array variables conservatively; we omit a detailed discussion on array types for simplicity.

*Access Expressions.*
Given a universe of object variables $V$, *access expressions* are constructed as follows:

(a) A variable $\texttt{ob}_i$ is an access expression of type $C_i$.
(b) Let $e_j$ be an access expression of non-primitive type $C_j$, and $f_k$ be a field of the class $C_j$ of type $C_k$. Then $\mathbf{e}:\ e_j.f_k$ is an access expression of type $C_k$.

Informally, access expressions are of the form $\texttt{ob}.f_1.f_2.\ldots.f_k$ for some valid sequence of field accesses $f_1,\ldots,f_k$. Let $\mathsf{Type}(e)$ denote the type of an access expression $e$. A *runtime environment* associates a set of concrete memory locations and values to each object instance and its fields.

*Aliasing, Sharing.*
Aliasing is a relationship between access expressions such that two access expressions $e_1$ and $e_2$ are aliased under runtime environment $R$, if they refer to the same object instance. Two objects $\texttt{ob}_i$, $\texttt{ob}_j$ are said to *share* in a runtime environment $R$ if some access expression of the form $\texttt{ob}_i.f_1\ldots.f_k$ aliases another expression of the form $\texttt{ob}_j.g_1\ldots.g_j$. In a type system similar to `Java`'s type system, we can generally assume that if $e_1$ and $e_2$ are aliased, then $C_1:\ \mathsf{Type}(e_1)$ is a subtype of $C_2:\ \mathsf{Type}(e_2)$ or vice-versa. In this case, we also assume that if $f_1,\ldots,f_k$ are the common fields between $C_1$ and $C_2$, then $e_1.f_i$ aliases $e_2.f_i$ for all $1\leq i\leq k$.

A method $m$ of class $\mathscr{C}$ is associated with a signature $sig(m)$ that defines the types for the formal parameters of $m$, and a return type. Each method $m$ is always executed on an object of some type $C_i$. The object on which the method is executed is referred to as "`this`" within the method body. The method body consists of a sequence of statements, including calls to other member methods of classes within $\mathscr{L}$. The operational semantics of $m$ are defined using a control-flow graph (denoted $cfg(m)$). We define $cfg(m)=(V_c,E_c,S)$, where $V_c$ is a set of program points, and $E_c$ is a set of edges, each labeled with a unique program statement $s\in S$.

## 2.1 Synchronization Primitives

*Lock-based synchronization.*
We seek to analyze object libraries that support concurrent accesses to their fields and methods. Therefore, we assume that synchronization statements for *lock-acquisition* and *lock-release* are used to provide *mutual exclusion* for shared data. We assume that these statements are of the form `lock(ob)` and `unlock(ob)`, where `ob` is some object variable. A thread executing `lock(ob)` is blocked unless it can successfully acquire the lock associated with `ob`. The statement `unlock(ob)` releases the lock, returning it to the unlocked state.

In certain languages such as `C++/pthread` and C# there is a designated type for lock variables. For instance, the `pthread` library uses the type `pthread_mutex_t` for mutex locks, and the functions `pthread_mutex_lock` and `pthread_mutex_unlock` to implement acquisition and release of mutexes. In such a model, the programmer decides upon a lock variable to protect access to one or more shared data items. It is the programmer's responsibility to ensure that each access to shared data is preceded by necessary lock acquisition and release, and that the locking discipline is uniform. Failure to do so results in low-level data races or high-level atomicity violations.

```
 1: public class Foo {
 2:     public void method1() {
 3:         . . .
 4:         synchronized (mon) {
 5:             . . .
 6:         }
 7:     }
 8:     public synchronized void method2 () {
 9:         . . .
10:     }
11: }
```

Fig. 2.1: Monitor Usage

*Monitor-based synchronization.*
Languages like `Java`, use *monitors* to implement synchronization[2]. A monitor object is a special object with built-in mutual exclusion and thread synchronization capabilities. A *monitored region* corresponding to the monitor `mon` is a sequence of statements that begins with the acquisition of `mon`, and ends with the release of `mon`. In `Java`, *any* object can be used as a monitor, and a monitored region is specified with the help of the `synchronized` keyword. For instance, in the example shown in Fig. 2.1, Lines 4-6 constitute a monitored region associated with the monitor object `mon`.

`Java` also allows using the keyword `synchronized` in a method signature (Line 8 in Fig. 2.1), which makes the entire method a monitored region corresponding to the implicit object (`this`) on which the method is invoked.

In theory, a monitor is associated with two explicit queues, an *entry queue* and a *wait queue*. In `Java`, instead of queues, each monitor maintains an *entry set* and a *wait set*. Queues are intended to implement FIFO access to a monitored region; `Java` makes no such guarantees. In the rest of the presentation, we assume the `Java` model, *i.e.*, we have an entry set and a wait set for each monitor. The entry set is used primarily for *mutual exclusion*, while both sets are used in concert for cooperative synchronization.

*Mutual Exclusion with Monitors.*
Let `mon` be a monitor object. When a thread $T$ reaches the beginning of a monitored region for `mon`, it is placed in the entry set of `mon`. $T$ is granted access to a monitored region if no other thread is executing inside it; we say that $T$ *acquires* `mon` when it enters the monitored region. Any other thread $T'$ that reaches the beginning of the monitored region once `mon` is acquired by $T$, is placed in the entry set for `mon`. Once $T$ leaves the monitored region, we say that $T$ *releases* `mon`. At this time, some (randomly chosen) thread in the entry set is able to acquire `mon`. Essentially, a monitor maintains the invariant that at any given time there is at most one thread inside the monitored region.

In effect, monitors mimic locks: replace the beginning of a monitored region (for `mon`) with `lock(mon)`, the end of the monitored region with `unlock(mon)`. One advantage with having a monitored region is that every monitor acquisition has a matching release, and monitors can be acquired and released only in a strictly nested fashion, *i.e.*, if `ob1` is acquired before `ob2`, then the monitors are released in the *reverse* order.

[2] With the addition of the `java.util.Concurrent` library to `Java`, there is now language support for an explicit lock construct.

| $P$: | $Q$: |
|---|---|
| `a0: synchronized (mon) {` | `b0: synchronized (mon) {` |
| `a1:` $A_1$`;` | `b1:` $B_1$`;` |
| `a2:     mon.wait();` | `b2:     mon.notify();` |
| `a3:` $A_2$`;` | `b3:` $B_2$`;` |
| `a4: }` | `b4: }` |

Fig. 2.2: Wait-Notify Monitors

*Co-operation with Monitors.*
Use of signaling allows monitors to implement cooperation between threads. We focus on the *wait-notify* style of monitors used by `Java`. Each monitor is provided with two special methods: `wait` and `notify`. We explain the semantics of `wait` and `notify` methods with the code fragment shown in Fig. 2.2.

A thread (say $T_1$) executing code fragment $P$ acquires the monitor `mon`, at Line `a0`. After executing code-block $A_1$, in Line `a2`, $T_1$ executes the `mon.wait()` statement, which has the following effect: (a) release the monitor `mon`, (b) add $T_1$ to the wait set for `mon`, (c) suspend execution of $T_1$. Assume that some other thread, say $T_2$, reaches the beginning of the monitored region in code fragment $Q$ (Line `b0`) after $T_1$ has executed Line `a0`. $T_2$ is then placed in the entry set for `mon`. Once $T_1$ releases `mon` in Line `a2`, $T_2$ can enter the monitored region, subsequently executing $B_1$, followed by `mon.notify()`. The effect of the notify statement is to remove any one thread (say $T_1$) from the wait set for `mon`, and place it in the entry set for `mon`. $T_1$ cannot resume execution as it is still in the entry set for `mon`, and $T_2$ "owns" `mon`. Once $T_2$ executes Lines `b3, b4`, it releases `mon`. Now $T_1$ may acquire `mon`, and proceed, executing Lines `a3, a4`.

*Condition Variables.*
To contrast with `wait-notify` monitors in `Java`, we briefly discuss signaling-based synchronization in the `pthread` library. A *condition variable* `cv` is a shared resource that is used in conjunction with a mutex lock `l`. The variable `cv` is typically associated with a Boolean-valued expression known as the condition. The method `pthread_cond_wait` has two arguments: `cv` and `l`, and its semantics are similar to that of `wait`: unlock mutex `l`, start waiting on variable `cv`, and upon being notified re-acquire mutex `l`. The method `pthread_cond_signal` takes one argument: `cv`, and its semantics are similar to that of `notify`: issue notification to the variable `cv`.

The discussion on primitives in other languages hopes to illustrate that the algorithmic underpinnings of our analysis techniques would remain largely unchanged while analyzing concurrent libraries written in other languages such as `C++/pthread` or C#. For instance, at the level of abstraction used in this paper, mutex locks in languages such as `C++/pthread` and the `synchronized` keyword in `Java` are similar. Also, `wait/notify`-based synchronization in `Java` and condition variables in `C++/pthread` also share the same structure. In rest of the paper, we focus on `Java` libraries, and present experimental results only pertaining to these libraries. In Sections 3-4, we consider monitors used only for ensuring mutual exclusion, and as such, use the term lock interchangeably with such a monitor. In Sections 5 and 6, we consider deadlocks arising from certain usage patterns for `wait` and `notify` methods.

**Procedure** computeLG($m$)

```
begin
    worklist := $\emptyset$
    V, E, lockset, roots := $\emptyset$
    worklist.push($\top_m$)
    while (worklist $\neq \emptyset$) do
        /* Edge $u \xrightarrow{s} v$ in $cfg(m)$ */
        $u$ := worklist.deque()
        $succs(u) = \{v | (u \xrightarrow{s} v) \in cfg(m)\}$
        foreach ($v$ in $succs(u)$) do
            $old\_sum$ := psum($v$)
            $new\_sum$ := computeFlow($u, s, v$)
            if ($old\_sum \neq new\_sum$) then worklist.push($v$)
            summary($v$) := psum($v$) $\sqcup$ $new\_flow$
    summary($m$) := psum($\perp_m$)
    summaries_map.put($m$, summary($m$))
end
```

## 3 Approach

In this section, we introduce the formal definitions for a lock-order graph, deadlockability analysis, and deadlock-causing aliasing patterns. We then discuss a scheme to encode a lock-order graph into a constraint to enable symbolic reasoning with an SMT-based constraint solver.

### 3.1 Static Computation of Lock-Graphs

*Lock-Order Graph.*
A lock-order graph for method $m$ denoted $lg(m)$ is a tuple $(V, E)$, where $V$ is a set of access expressions, and $E$ is a set of edges. An edge $e_1 \rightarrow e_2$ denotes a pair of nested lock statements `lock(x)` followed by `lock(y)` wherein `x` aliases the access expression $e_1$, `y` aliases the access expression $e_2$, and the lock acquisitions are nested along some path in $cfg(m)$ or along a path in the $cfg$ of one of $m$'s callees. In what follows, we frequently use the shorter term lock-graph interchangeably with lock-order graph.

*Static Forward Lock-Graph Analysis.*
Interprocedural lock-order graph computation for the methods of a given library involves summarization of each method $m$ within the library. Let $u \xrightarrow{s} v$ be an edge in $cfg(m)$. The partial summary of $m$ at control-flow node $v$ (denoted `psum`$(v)$) is the symbolic state of $m$ after executing the statement $s$. It is described as the data structure $(lg(V,E), \texttt{lockset}, \texttt{roots})$, where $lg(V,E)$ is the lock-order graph, `lockset` is the set of locks acquired (but not released) by $m$ at $v$, `roots` is the set of locks that do not have any incoming edges[3].

We closely follow the technique described in [30] for fixpoint-based summary computation. The procedure computeLG implements a simple work-list based forward flow analysis.

[3] In the actual implementation, we also track as a part of the summary a mapping *env*, that tracks any local variables that may be aliased to global variables on the heap, and thus escape the scope of $m$. We omit this for simplicity.

**Function** `computeFlow`

```
   input  : cfg edge: $u \xrightarrow{s} v$
   output: partial summary $out$
1  begin
2      $in := \mathtt{psum}(u)$, $out := \emptyset$
       /* $in$ = ($\mathtt{V}_i$,$\mathtt{E}_i$,$\mathtt{roots}_i$,$\mathtt{lockset}_i$), $out$ = ($\mathtt{V}_o$,$\mathtt{E}_o$,$\mathtt{roots}_o$,$\mathtt{lockset}_o$)   */
3      switch (s) do
           /* Lock acquisition.                                                */
4          case $\mathtt{lock}(\mathtt{mon})$ :
5              foreach ($\mathtt{mon}' \in \mathtt{lockset}_i$) do $\mathtt{E}_o := \mathtt{E}_i \cup \{(\mathtt{mon}',\mathtt{mon})\}$
6              $\mathtt{lockset}_o := \mathtt{lockset}_i \cup \{\mathtt{mon}\}$
7              if ($\mathtt{roots}_i = \emptyset$) then $\mathtt{roots}_o := \{\mathtt{mon}\}$
           /* Lock release.                                                    */
8          case $\mathtt{unlock}(\mathtt{mon})$ :
9              $\mathtt{lockset}_o := \mathtt{lockset}_i - \{\mathtt{mon}\}$
10             if ($\mathtt{lockset}_o = \emptyset$) then $\mathtt{roots}_o := \emptyset$
           /* Method Invocation.                                               */
11         case $\mathtt{m}'(a_1,\ldots,a_k)$ :
               /* Check if $\mathtt{summary}(m') = (\mathtt{V}',\mathtt{E}',\mathtt{lockset}',\mathtt{roots}')$ exists, if not
                  compute it.                                                  */
12             if ($\mathtt{summaries\_map}$.contains($m'$)) then $\mathtt{summary}(m') := \mathtt{summaries\_map}$.get($m'$)
13             else $\mathtt{summary}(m') := \mathsf{computeLG}(m')$
               /* Map formal parameters in the summary to actual parameters.   */
14             $sum' := \mathtt{summary}(m')|_{\forall i: f_i \mapsto a_i}$
               /* Concatenate new summary with current summary.                */
15             $\mathtt{E}_o := \mathtt{E}_i \cup \mathtt{E}'$, $\mathtt{V}_o := \mathtt{V}_i \cup \mathtt{V}'$
16             foreach ($\mathtt{mon} \in \mathtt{lockset}$) do
17                 foreach ($\mathtt{mon}' \in \mathtt{roots}'$) do
18                     $\mathtt{E}_o := \mathtt{E}_o \cup \{(\mathtt{mon},\mathtt{mon}')\}$
19         otherwise
20             $out := in$
21 end
```

We introduce a single (dummy) entry-point $\top_m$ that has all the actual entry-points of the $m$ as successors, and an exit-point $\bot_m$ that is the successor of all actual exit-points (*i.e.* `return` statements) for $m$. We assume that the partially computed summary at each point in the control-flow graph is initialized to $(\emptyset,\emptyset,\emptyset)$. In each step, a new edge $(u \xrightarrow{s} v)$ in cfg(m) is examined. Using the flow equations for the edge as specified by the function computeFlow and the partial summary at control point $u$ ($\mathtt{psum}(u)$), we obtain $\mathtt{psum}(v)$ (Line 10). If the new $\mathtt{psum}(v)$ is different from the original $\mathtt{psum}(v)$ , then $v$ is added to the work-list (Line 11), and the new $\mathtt{psum}(v)$ is *merged* with the old (Line 12). The merge operation ($\sqcup$) computes the union of each component in the partial summary. Finally, the summary of method $m$ (and $lg(m)$ contained therein) is obtained as the merge of the partial summaries at $\bot_m$, which is then stored into a map (Lines 13-14).

We remark that in the presence of recursive types (classes containing themselves as members, for instance), and recursion/loops in the CFG, the fixpoint computation may not terminate, in general. We ensure termination by artificially bounding the size of the access expressions allowed as nodes in the lock graphs.

The computeFlow function is used to compute the effect of a statement $s$ on the partial summary. For the edge $u \xrightarrow{\mathtt{lock(mon)}} v$ corresponding to a lock-acquisition, we add edges from every lock $\mathtt{mon}'$ in $\mathtt{lockset}(u)$ to mon, and then add mon to $\mathtt{lockset}(v)$ (Lines 5-7). The edge $u \xrightarrow{\mathtt{unlock(mon)}} v$ corresponding to a lock-release is modeled by setting $\mathtt{lockset}(v)$ to the set obtained by removing mon from $\mathtt{lockset}(u)$ (Lines 9-10). Upon encountering a call to a method $m'$, we check if the summary for $m'$ has been computed, and if not, we first compute it. We then replace the formal parameters in $\mathtt{summary}(m')$ with the actual parameters at the call-site, and concatenate the result $sum'$ with the partial summary computed thus far (Lines 12-16). Concatenation involves adding edges from every lock in lockset to every root in the $\mathtt{roots}'$ (in $sum'$) and adding all other edges in $lg'$ (in $sum'$) to the lock-graph in the partial summary.

## 3.2 Deadlockability

Let $m_1,\ldots,m_k$ be a set of methods in library $\mathscr{L}$ that are concurrently invoked by $k$ separate threads. For ease of exposition, we consider the case of two threads (i.e., $k = 2$). However, our results readily extend to *arbitrary* values of $k$. Let objects $\mathtt{ob}_1,\ldots,\mathtt{ob}_k$ denote a set of objects on which the methods $m_1,\ldots,m_k$ are invoked. Let $\mathtt{ob}_{k+1},\ldots,\mathtt{ob}_n$ be the set of parameters to these method calls. Let $lg(m_1)$ and $lg(m_2)$ be the lock order graphs for the methods $m_1$ and $m_2$ after substituting the this object and the formal parameters in $m_1$ and $m_2$ with $\mathtt{ob}_1,\ldots,\mathtt{ob}_n$. We assume that $lg(m_1)$ and $lg(m_2)$ are themselves cycle free[4]. Let $V_{1,2} = \{\mathtt{e}_1,\ldots,\mathtt{e}_m\}$ denote the set of access expressions occurring in $lg(m_1)$ or $lg(m_2)$. We first characterize the patterns of aliasing/sharing between the access expressions corresponding to $\mathtt{ob}_1,\ldots,\mathtt{ob}_n$ under some fixed runtime environment $R$.

**Definition 3.1 (Aliasing Pattern)** An aliasing pattern $\alpha$ over a set of access expressions $V$ is a symmetric, reflexive and transitive relation over $V$. If $(e_1,e_2) \in \alpha$ then $(e_1.f_i, e_2.f_i) \in \alpha$ for all shared fields $f_i$ between $\mathsf{Type}(e_1)$ and $\mathsf{Type}(e_2)$.

Given graphs $G_1(V_1,E_1)$ and $G_2(V_2,E_2)$, we use $G = G_1 \sqcup G_2$ to denote the union of the two graphs (i.e. the set of vertices of $G$ is $V_1 \cup V_2$, and the set of edges is $E_1 \cup E_2$). For lock-graphs $lg(m_1)$ and $lg(m_2)$, we refer to $lg(m_1) \sqcup lg(m_2)$ as the merged lock-graph for $m_1$ and $m_2$. Given an aliasing pattern $\alpha$ over the nodes of a merged graph $lg(m_1) \sqcup lg(m_2)$, we *fuse* the nodes $e_i, e_j$ of the graph if $(e_i, e_j) \in \alpha$. The outgoing and incoming edges to the individual nodes $e_i, e_j$ are preserved by the fused node. Let $\alpha \rhd G$ denote the resulting graph after merging all aliased nodes.

**Definition 3.2 (Deadlock Causing Pattern)** An aliasing pattern $\alpha$ is potentially *deadlock-causing* for $m_1, m_2$ iff $\alpha \rhd (lg(m_1) \sqcup lg(m_2))$ contains a cycle. An aliasing pattern that is not deadlock-causing is termed *safe*.

**Example 3.1** Consider two methods from the java.awt.EventQueue class: $m_1$ (wakeup) and $m_2$ (postEventPrivate), shown in Fig. 1.1. Sec. 1 illustrates the individual lock-order graphs $lg(m_1)$ and $lg(m_2)$. Following the notation established, let $\mathtt{ob}_1, \mathtt{ob}_2$ denote the objects on which methods $m_1, m_2$ are invoked, respectively. The access expressions involved in the lock graph $G : lg(m_1) \sqcup lg(m_2)$ are $V_{1,2} = \{\mathtt{ob}_1, \mathtt{ob}_2, \mathtt{ob}_1.\mathtt{nextQueue}, \mathtt{ob}_2.\mathtt{nextQueue}\}$. Let $\alpha_1$ be the aliasing pattern $\{(\mathtt{ob}_1, \mathtt{ob}_2.\mathtt{nextQueue})\}$. The merged lock graph $\alpha_1 \rhd G$ is shown below:

[4] This assumption relies on re-entrancy of locks. Java monitors are re-entrant. Mutexes in C/C++ with the pthread library are commonly defined to be re-entrant. Thus this assumption generally holds true for the libraries that we seek to analyze.

$\alpha_1 \triangleright G$:

$$\begin{tikzcd}\texttt{ob}_2 \arrow[r, "1"] & \texttt{ob}_2\texttt{.nextQueue}, \texttt{ob}_1 \arrow[r, "2"] & \texttt{ob}_1\texttt{.nextQueue}\end{tikzcd}$$

The pattern $\alpha_1$ does not cause a deadlock. However, the following pattern $\alpha_2$, considered in Sec. 1 is deadlock-causing:

$$\alpha_2 : \{(\texttt{ob}_2, \texttt{ob}_1\texttt{.nextQueue}), (\texttt{ob}_1, \texttt{ob}_2\texttt{.nextQueue})\}$$

**Definition 3.3 (Deadlockable)** A library is termed potentially *deadlockable* if there exists a pair of methods $m_1, m_2$, and some aliasing pattern $\alpha$ amongst the access expressions in $V_{1,2}$ such that $\alpha \triangleright (lg(m_1) \sqcup lg(m_2))$ contains a cycle.

A simplistic approach consists of (a) enumerating all possible aliasing patterns $\alpha$, and (b) checking every graph $\alpha \triangleright G$ for a cycle. As pointed out in [30], there may exist a huge number of aliasing/sharing relationships between the parameters, invoked objects, and their fields. Explicit reasoning over such a large number of patterns is intractable. Hence, we use a symbolic representation to encode the graphs and the aliasing patterns as constraints, enabling the use of SAT-Modulo Theory (SMT) solvers to perform the enumeration efficiently.

## 3.3 Symbolic Encoding

We first discuss how we can encode the cycle detection problem into an efficient theory amenable to an SMT solver. The inputs to this problem are a graph $G = lg_1 \sqcup lg_2$, and a fixed aliasing pattern $\alpha$. In Sec. 4 we will use this encoding to efficiently enumerate all possible patterns to detect potential deadlocks and derive interface contracts.

The overall strategy consists of two parts: We first encode a lock graph $G$ over a set of access expressions $V_G$ as a logical formula $\Psi(G)$. Next, we show how a given alias pattern $\alpha$ may be encoded as a formula $\Psi(\alpha)$. As a result, we guarantee that $\Psi(\alpha) \wedge \Psi(G)$ is unsatisfiable if and only if $\alpha \triangleright G$ has a cycle. The formula $\Psi(G)$ represents a topological ordering of the graph and $\Psi(\alpha)$ places equality constraints on the vertex numbers based on aliasing. If the result is unsatisfiable then no topological order can exist, indicating a cycle.

*Graph Encoding.*
Corresponding to each node $v_i \in V$, we create an integer variable $x(v_i)$ representing its rank in a topological ordering of the node $v_i$. Corresponding to each edge $v_i \to v_j$ in the graph, we add the constraint $x(v_i) < x(v_j)$. The resulting formula $\Psi(G)$ is the conjunction of all edge inequalities:

$$\Psi(G) : \left[ \bigwedge_{(v_i,v_j)\in E} \big(\mathbf{x}(\mathbf{v_i}) < \mathbf{x}(\mathbf{v_j})\big) \right] . \tag{3.1}$$

**Example 3.2** Consider once again the running example from Fig. 1.1, continuing with the notation established in Ex. 3.1. The merged lock graph $G :\ lg(m_1) \sqcup lg(m_2)$ is recalled in Fig. 1.2. The constraint $\Psi(G)$ for this graph is as follows:

$$(x(\texttt{ob}_1) < x(\texttt{ob}_1\texttt{.nextQueue})) \ \wedge\ (x(\texttt{ob}_2) < x(\texttt{ob}_2\texttt{.nextQueue}))\,.$$

*Aliasing Pattern Encoding.*
Given an aliasing pattern $\alpha$, we wish to derive a formula $\Psi(\alpha, G)$ whose satisfiability indicates the absence of a cycle in $\alpha \triangleright G$ (and conversely). This is achieved by encoding $\alpha$ by means of a set of equalities as follows:

$$\Psi(\alpha) : \left[ \bigwedge_{(\mathtt{e}_i, \mathtt{e}_j) \in \alpha} \left( \mathbf{x}(\mathtt{e_i}) = \mathbf{x}(\mathtt{e_j}) \right) \right] . \tag{3.2}$$

In effect, the rank of the access expressions that are aliased is required to be the same in the topological order.

**Example 3.3** Continuing with Ex. 3.2, the aliasing pattern $\alpha_1 : \; \{(\mathtt{ob}_2, \mathtt{ob}_1.\mathtt{nextQueue})\}$ may be encoded as: $\Psi(\alpha_1) : (x(\mathtt{ob}_2) = x(\mathtt{ob}_1.\mathtt{nextQueue}))$.

Given an aliasing pattern $\alpha$, and a graph $G$, the formulae $\Psi(G)$, $\Psi(\alpha)$ are conjoined into a single formula $\Psi(\alpha, G) : \Psi(G) \wedge \Psi(\alpha)$ that enforces the requirements for a topological order specified by $G$, as well as for merging nodes according to the aliasing pattern $\alpha$.

**Example 3.4** Continuing with Ex. 3.3, recall $\Psi(G)$ from Ex. 3.1, consider the combined formula:

$$\begin{aligned} \Psi(\alpha_1, G) : \; & (x(\mathtt{ob}_1) < x(\mathtt{ob}_1.\mathtt{nextQueue})) \;\wedge\; (x(\mathtt{ob}_2) < x(\mathtt{ob}_2.\mathtt{nextQueue})) \;\wedge \\ & (x(\mathtt{ob}_2) = x(\mathtt{ob}_1.\mathtt{nextQueue})) \end{aligned}$$

This formula is satisfiable in the theory of integers, indicating a topological ordering over $\Psi : \alpha_1 \triangleright G$, thus showing that no cycle exists in $\alpha_1 \triangleright G$. On the other hand, consider the formula $\Psi(\alpha_2, G)$ obtained from the pattern:

$$\begin{aligned} & \alpha_2 : \{(\mathtt{ob}_2, \mathtt{ob}_1.\mathtt{nextQueue}), (\mathtt{ob}_1, \mathtt{ob}_2.\mathtt{nextQueue})\} \\ \textit{i.e.}, \;\; & \Psi(\alpha_2) : (x(\mathtt{ob}_2) = x(\mathtt{ob}_1.\mathtt{nextQueue})) \;\wedge\; (x(\mathtt{ob}_1) = x(\mathtt{ob}_2.\mathtt{nextQueue})) \end{aligned}$$

The combination of $\Psi(G) \wedge \Psi(\alpha_2)$ is clearly unsatisfiable indicating that $\alpha_2 \triangleright G$ has a cycle, which in turn shows that $\alpha_2$ may cause a deadlock.

**Theorem 3.1** *The formula $\Psi(\alpha, G)$ is* satisfiable *iff $\alpha \triangleright G$ does not have a cycle.*

*Proof* We begin by simplifying the statement of the theorem. Let $G' = \alpha \triangleright G$. Let $\Psi(G')$ be the encoding for $G'$ as per (3.1). We observe that $\Psi(G')$ can be obtained from $\Psi(\alpha, G)$ by by replacing integer variables $x(v_i)$ and $x(v_j)$ by a new variable $x_{ij}$, if $\Psi(\alpha)$ contains the relation $x(v_i) = x(v_j)$. Note that $\Psi(G')$ is satisfiable iff $\Psi(\alpha, G)$ is satisfiable. Thus, we now wish to prove that $\Psi(G')$ is satisfiable iff $G'$ is acyclic.

We first prove that if $G'$ is acyclic, $\Psi(G')$ obtained as per (3.1) is satisfiable. Note that the edge relation of an acyclic graph $G'(V, E)$ defines a strict partial order[5] on the set of its vertices, and for a given strict partial order $(E)$ we can define the linear extension of $E$ (denoted $E_{tot}$) by the order-extension principle. By definition, $E_{tot}$ is a total order, and if $(u, v) \in E$, then $(u, v) \in E_{tot}$. Since $E_{tot}$ is a total order, we can define a bijection $f$ from the set of vertices $V$ to $\mathbb{N}$ such that if $(u, v) \in E_{tot}$, $f(u) < f(v)$. Thus, the interpretation of $\Psi(G')$ where each $x(u)$ is replaced by $f(u)$ evaluates to *true*, *i.e.*, $\Psi(G')$ is satisfiable.

[5] A strict partial order is a transitive and asymmetric binary relation on a set.

To prove the reverse direction, we prove by contradiction. Assume that $\Psi(G')$ is satisfiable and $G'(V,E)$ contains a cycle. Since $\Psi(G')$ is satisfiable, we can find a satisfying assignment to $\Psi(G')$ such that each $x(v_i)$ corresponds to a distinct integer. By definition, each constraint $x(v_i) < x(v_j)$ corresponds to an edge $(v_i, v_j) \in E$. Since $G'$ contains a cycle, it contains a path $\pi = (v_1, \ldots, v_k, v_1)$ in $G'$, s.t. each consecutive pair of vertices in $\pi$ is in $E$. The conjunction of constraints corresponding to $\pi$ contains the inequality $(x(v_k) < x(v_1))$ and by transitivity of $<$ over integers, also contains $(x(v_1) < x(v_k))$. This is a contradiction as each $x(v_i)$ is a distinct integer. $\square$

*Constraint Solving.*
Given an aliasing pattern $\alpha$, the constraint $\Psi(\alpha)$ is a conjunction of equalities, whereas $\Psi(G)$ is a conjunction of inequalities of the form: $v_i < v_j$, i.e., *unit two variable per inequality* (UTVPI) constraint [18]. In practice, Boolean combinations of UTVPI and equality constraints can be solved quite efficiently using modern SMT solvers such as Yices and Z3 [11, 22].

We also note that the problem of solving a set of UTVPI constraints is equivalent to cycle detection in a graph. Therefore, our reduction in this section has not gained/lost in complexity. On the other hand, encoding the graph cycle detection problem as a UTVPI constraint in an SMT framework allows us to efficiently make use of strategies such as *incremental cycle detection* and *unsatisfiable cores*. The subsequent section shows the use of these primitives to effectively enumerate all aliasing patterns by computing *subsumed* and *subsuming* patterns. The discovery of such patterns reduces the set of aliases to be examined and speeds up our approach enormously.

## 4 Aliasing Pattern Enumeration

We now consider the problem of enumerating all possible aliasing patterns, in order to generate the interface contracts. The number of such patterns is exponential in the number of nodes of the lock-order graphs. Following Sec. 3, we need to enumerate all possible equivalence classes over the sets of nodes in the lock-order graphs. A naive approach thus suffers from an exponential blow-up. We avoid this using various key optimizations:

(a) We prune the lock-order graphs to remove all nodes that cannot contribute to a potential deadlock.
(b) We restrict the possible aliasing patterns with the help of a prior alias analysis and typing rules imposed by the underlying programming language.
(c) Based on the set of aliasing patterns already enumerated, we remove sets of *subsumed* or *subsuming* aliasing patterns from consideration.

*Graph Pruning.*
Let $E_i$, $V_i$ represent the edges and vertices of the lock graph $G_i :\ lg(m_i)$. This pruning strategy is based on the observation that nested lock acquisitions are relatively uncommon and non-nested lock acquisitions may be removed from the lock graph. As a results, nodes without any successors and predecessors can be trivially removed. This results in a large reduction in the size.

A *terminal node* in the graph is defined as one without any successors. Similarly a node in the lock graph is termed *initial* if it has no predecessors. In general, a terminal node $\mathsf{e}_i \in V$ *cannot* be removed without missing any potential deadlocks.

**Example 4.1** Returning to the lock graph in Fig. 1.2 we note that the two terminal nodes may not be removed since their incoming edges can be used in a potential cycle. The same consideration applies to initial nodes.

However, a terminal node *can* be removed if all the other nodes to which it *may alias* to are also terminal. Similarly, an initial node can be removed if all the other nodes to which it may alias are also initial. The pruning strategy for removing terminal/initial nodes of the graph utilizes the result of a conservative may-alias analysis. Let $\mathsf{mayAlias}(v) = \{u \in V \mid u \text{ may-alias } v\}$.

1. Let $v$ be a terminal node such that all nodes in $\mathsf{mayAlias}(v)$ are also terminal. We remove the vertices in $\mathsf{mayAlias}(v)$ from the graph.
2. Let $u$ be an initial node such that all nodes in $\mathsf{mayAlias}(u)$ are also initial. We remove all nodes in $\mathsf{mayAlias}(u)$ from the graph.

The removal of a terminal/initial node from the graph may create other terminal/initial nodes respectively. Hence, we iterate steps 1 and 2 until no new nodes can be removed. The mayAlias relationship can be safely approximated in languages like `Java` by *type-masking*. As a result, we regard two nodes as aliased for the purposes of lock graph pruning, if the types of their associated access expressions are compatible (one is a sub-type of another). Note that no potential deadlocks are lost in this process. Our experiments indicate that the pruned lock graph is an order of magnitude smaller than the original graph obtained from static analysis, making this an important step in making the overall approach scalable. We now shift our focus to reducing the number of aliasing patterns to be enumerated.

*Reducing Aliasing Patterns.*
Given two graphs with $n$ nodes each, the number of possible aliasing patterns that need to be considered across the nodes of the two graphs is exponential in $n$. In our experiments with `Java` libraries, we have observed that the extensive use of locking with the `synchronized` keyword gives rise to lock-order graphs containing 100s of nodes, which are reduced to lock-order graphs with 10s of nodes after pruning. However, given the exponential number of aliasing patterns that may exist, we need to impose restrictions on the set of aliasing patterns that we examine. First of all, it suffices to consider aliasing patterns that respect the type safety considerations of the language and the conservative may-alias relationships between nodes.

**Definition 4.1 (Admissible)** An aliasing pattern $\alpha$ is admissible iff for all $(u,v) \in \alpha$, $u \in \mathsf{mayAlias}(v)$. Once again, type information can be used in lieu of alias information for languages such as `Java`.

Another important consideration for reducing the aliasing patterns, is that of *subsumption*. Subsumption is based on the observation that for a deadlock causing pattern $\alpha$ adding more aliases to $\alpha$ does not remove the deadlock. Similarly, for a safe pattern $\beta$, removing aliases from $\beta$ does not cause a deadlock.

**Definition 4.2 (Subsumption)** A pattern $\alpha_2$ *subsumes* $\alpha_1$, denoted $\alpha_1 \subseteq \alpha_2$, iff $\forall (u,v) : (u,v) \in \alpha_1 \Rightarrow (u,v) \in \alpha_2$. In other words, $\alpha_1$ is a sub-relation of $\alpha_2$.

**Lemma 4.1** *If $\alpha_1, \alpha_2$ are aliasing patterns, and $\alpha_1 \subseteq \alpha_2$, then the following are true:*

*(A) $\alpha_1$ is deadlock-causing $\Rightarrow$ $\alpha_2$ is deadlock-causing,*
*(B) $\alpha_2$ is safe $\Rightarrow$ $\alpha_1$ is safe.*

*Proof* As $\alpha_1 \subseteq \alpha_2$, $\Psi(\alpha_2, G)$ can be expressed as $\Psi(\alpha_1, G) \wedge \Psi(\alpha_2 \setminus \alpha_1)$. Since we know that $\alpha_1$ is deadlock-causing, by definition, $\Psi(\alpha_1, G)$ is unsatisfiable, implying that $\Psi(\alpha_2, G)$ is unsatisfiable. □

Note that (B) is simply the contrapositive of (A) and is stated here in Lemma 4.1 for the sake of exposition.

**Definition 4.3 (Maximally Safe/Minimally Unsafe)** A pattern $\alpha$ that causes a deadlock is *minimally unsafe* iff for any $(u,v) \in \alpha$, $\alpha - \{(u,v)\}$ is not deadlock causing. Similarly, a safe (non-deadlock) pattern $\alpha$ is maximally safe if, for any $(u,v) \notin \alpha$, $\alpha \cup \{(u,v)\}$ is deadlock causing.

Following Lemma 4.1, it suffices to enumerate only the maximally safe and minimally unsafe patterns. Hence, after enumerating a pattern $\alpha$ that is *safe*, we can add previously unaliased pairs of aliases to $\alpha$ as long as the addition does not cause a deadlock. The resulting pattern is a *maximally safe pattern*. Similarly, upon encountering a deadlock-causing pattern $\beta$, we remove "unnecessary" alias pairs from $\beta$ as long as pairs that contribute to some cycle in $\beta \triangleright G$ can be retained.

**Example 4.2** Consider the aliasing pattern $\alpha_0 : \emptyset$ for the example described in Sec. 1.1. Fig. 1.2 shows the resulting graph. We can add the pair $(\texttt{ob}_1, \texttt{ob}_2\texttt{.nextQueue})$ to $\alpha_0$ without creating any cycles. The resulting pattern $\alpha_1$ is shown in Ex. 3.1. However, if we add the pair $(\texttt{ob}_2, \texttt{ob}_1\texttt{.nextQueue})$ to $\alpha_1$ then we obtain a cycle in the graph. As a result, the pattern $\alpha_1$ is maximally safe.

The explicit enumeration algorithm (Algorithm 3) for aliasing patterns maintains a set $U$ of unexplored patterns, sequentially exhausting the unexamined patterns from this set while updating the set $U$. The algorithm terminates when $U = \emptyset$. First of all, a previously unexamined pattern $\alpha$ is chosen from the set $U$ (Line 4), and the graph $\alpha \triangleright G$ is examined for a cycle (Line 5). If the graph is acyclic, we keep adding previously unaliased pairs $(u,v)$ to $\alpha$ as long as the addition does not create a cycle in $\alpha' \triangleright G$, where $\alpha'$ is the symmetric and transitive closure of $\alpha \cup \{(u,v)\}$. The result is a pattern $\alpha$ that is maximally safe, which is then added to the set $\mathscr{S}$ (Line 9). We then remove all patterns $\beta$ that are subsumed by $\alpha$ from the graph $G$, as they are safe (Line 10). On the other hand, if the graph $\alpha \triangleright G$ has cycles, we choose some cycle $C$ in the graph (Line 12), and the aliases in $\alpha$ that involve the merged nodes in $C$. Discarding all the *superfluous* aliases not involving nodes in the cycle $C$ yields an alias relationship $\alpha' \subseteq \alpha$ that is still deadlock-causing [6] (Line 13). The set $U$ of unexamined patterns is pruned by removing all patterns that subsume $\alpha'$ (such patterns also cause a deadlock) (Line 14).

The application of Algo. 3 on the graph from Fig. 1.2 enumerates the max. safe/ min. unsafe patterns in Table 4.1.

*Symbolic Enumeration Algorithm.*
Algorithm 3 relies on explicit representation of the set $U$ of alias patterns in order to perform the enumeration. Representing an arbitrary set of relations explicitly is not efficient in

[6] Note that $\alpha'$ may not be a minimally unsafe relation.

**Algorithm 3**: EnumerateAllAliasingPatterns

```
   Input: $G$ : Graph
   Result: $\mathscr{D}$ : Deadlock Scenarios
1  begin
2      $U :=$ all legal aliasing patterns
3      while $U \neq \emptyset$ do
4          Choose element $\alpha \in U$.
5          if $\alpha \triangleright G$ is acyclic then
               /* Add aliases without creating a cycle */
6              foreach $(u,v) \notin \alpha$ do
                   /* Add $(u,v)$ and compute closure. */
7                  $\alpha' := \mathsf{Closure}(\{(u,v)\} \cup \alpha)$
8                  if $\alpha' \triangleright G$ is acyclic then $\alpha := \alpha'$
               /* $\alpha$ maximally safe */
9              $\mathscr{S} := \mathscr{S} \cup \{\alpha\}$
10             $U := U - \{\beta \mid \beta \subseteq \alpha\}$
11         else                                    /* $\alpha \triangleright G$ has a cycle */
               /* Choose a cycle $C$ */
12             $C :=$ FindACycle($\alpha \triangleright G$)
               /* Remove aliases that do not contribute to $C$ */
13             $\alpha' := \alpha \cap \{(u,v) \mid u,v \in C\}$
               /* $\alpha'$ is unsafe */
14             $U := U - \{\beta \mid \alpha' \subseteq \beta\}$
15             $\mathscr{D} := \mathscr{D} \cup \{\alpha'\}$
16
17 end
```

Table 4.1: Max. Safe/Min. Unsafe Patterns Enumerated.

| | |
|---|---|
| $\{(\mathtt{ob}_1, \mathtt{ob}_2), (\mathtt{ob}_1.\mathtt{nextQueue}, \mathtt{ob}_2.\mathtt{nextQueue})\}$ | SAFE |
| $\{(\mathtt{ob}_1, \mathtt{ob}_2.\mathtt{nextQueue})\}$ | SAFE |
| $\{(\mathtt{ob}_2, \mathtt{ob}_1.\mathtt{nextQueue})\}$ | SAFE |
| $\{(\mathtt{ob}_1, \mathtt{ob}_2.\mathtt{nextQueue}), (\mathtt{ob}_2, \mathtt{ob}_1.\mathtt{nextQueue})\}$ | DL |

practice. Therefore, we leverage the power of symbolic solvers to encode aliasing patterns succinctly. Specifically, we wish to represent the set $U$ of unexamined aliasing patterns with the help of a logical formula. Let $V = \{\mathsf{e}_1, \ldots, \mathsf{e}_k\}$ be the set of access expressions labeling the nodes of the graph $G$. We introduce a set of integer variables $y(\mathsf{e}_i)$, such that each $y(\mathsf{e}_i)$ corresponds to an access expression $\mathsf{e}_i$. We then encode all aliasing patterns with the help of a logical formula $\Psi_0$ involving the $y(\mathsf{e}_i)$ variables, as follows:

$$\Psi_0(V) = \forall_{e_i,e_j \in V} \left[ \begin{array}{ll} \bigwedge_{e_i \notin \mathsf{mayAlias}(e_j)} & (y(e_i) \neq y(e_j)) \wedge \\ \bigwedge_{e_i.f, e_j.f \in V} & ((y(e_i) = y(e_j)) \Rightarrow (y(e_i.f) = y(e_j.f))) \end{array} \right] \tag{4.1}$$

The formula $\Psi_0$, ensures the consistency of alias patterns considered in the enumeration process. Specifically, expressions that cannot be aliased to each other according to a conservative pointer analysis are not considered aliased in any of the patterns generated. Secondly, if $e_1, e_2$ are aliased then for every field $f$, $e_1.f$ and $e_2.f$ must be aliased (provided the two expressions are in the set $V$). Algorithm 4 shows the symbolic version of Algorithm 3. The correspondence between the two algorithms is immediately observable upon

**Algorithm 4**: SymbolicEnumerateAllAliasingPatterns

```
   Input: $G$ : Graph
   Result: $\mathscr{D}$ : Deadlock Scenarios
1  begin
2      $\Psi_U := \Psi_0(V)$ (encoding all alias patterns)
3      while $\Psi_U$ SAT do
4          $(y(\mathsf{e}_1),\ldots,y(\mathsf{e}_k)) :=$ Solution of $\Psi_U$.
5          $\alpha := \{(\mathsf{e}_i,\mathsf{e}_j) \mid y(\mathsf{e}_i) = y(\mathsf{e}_j)\}$.
           /* Construct $\Psi(\alpha,G)$ */
6          if $\Psi(\alpha,G)$ SAT then
               /* Add aliases without creating a cycle */
7              foreach $(\mathsf{e}_i,\mathsf{e}_j) \notin \alpha$ do
8                  $\alpha' := \mathsf{Closure}(\alpha \cup (\mathsf{e}_i,\mathsf{e}_j))$
9                  $\Psi(\alpha',G) := \Psi(\alpha,G) \wedge (x(\mathsf{e}_i) = x(\mathsf{e}_j))$
10                 if $\Psi(\alpha',G)$ SAT then $\alpha := \alpha'$
               /* $\alpha$ is maximally safe */
11             $\Psi_U := \Psi_U \wedge \bigvee_{(\mathsf{e}_i,\mathsf{e}_j)\notin\alpha} y(\mathsf{e}_i) = y(\mathsf{e}_j)$
12             $\mathscr{S} := \mathscr{S} \cup \{\alpha\}$
13         else
               /* $\Psi(\alpha,G)$ UNSAT */
14             $C :=$ MinUnsatCore$(\Psi(\alpha,G))$
15             $\alpha' := \{(\mathsf{e}_i,\mathsf{e}_j) \mid x(\mathsf{e}_i) < x(\mathsf{e}_j)$ constraint in $C\}$
               /* $\alpha'$ is unsafe */
16             $\Psi_U := \Psi_U \wedge \bigvee_{(\mathsf{e}_i,\mathsf{e}_j)\in\alpha'} y(\mathsf{e}_i) \neq y(\mathsf{e}_j)$
17             $\mathscr{D} := \mathscr{D} \cup \{\alpha'\}$
18 end
```

comparing them. Since we represent sets of aliasing patterns as a logical formula, a witness to the satisfiability of this formula is an aliasing pattern $\alpha$ (Line 5).

Recall from Sec. 3.3 that we can encode the problem of cycle detection in a graph using inequality constraints. Thus, in Line 6 we check the inequality constraints specified by the graph $G$ (*i.e.* $\Psi(G)$) conjoined with the previously unexamined aliasing pattern $\alpha$ (encoded as $\Psi(\alpha)$) for satisfiability. Satisfiability of this formula indicates that the graph $G$ is cycle-free, and we proceed to compute a maximally safe aliasing pattern from the given $\alpha$ (Line 7). Once a maximally safe $\alpha$ is obtained, we remove all aliasing patterns that are subsumed by $\alpha$ from the set of all aliasing patterns (represented by $\Psi_U$), and add $\alpha$ to $\mathscr{S}$ (Line 12). If the formula is unsatisfiable, then we obtain the minimal unsatisfiable core (Line 14) and extract the minimally unsafe aliasing pattern $\alpha'$ from the constraints represented in this core. We then remove all aliasing patterns that subsume $\alpha'$ from $\Psi_U$ (Line 16), and add the minimally unsafe $\alpha'$ obtained (if any) to the set $\mathscr{D}$ (Line 17).

Such a symbolic encoding of sets of aliasing patterns has many advantages, including: a) the power of constraints to represent sets of states compactly, and b) the use of blocking clauses to remove a set of subsumed/subsuming aliasing patterns. Modern UTVPI solvers such as Yices and Z3 incorporate techniques for fast and incremental cycle detection upon addition or deletion of constraints [11, 22]. This is very useful in the context of Algorithm 4. In practice, our use of subsumption and pruning ensures that a very small fraction amongst the alias patterns is explored by the symbolic algorithm.

*Deriving a Contract.*
The enumeration scheme in Algo. 3 and Algo. 4 can generate a contract that *succinctly*

represents the set of all safe aliasing patterns. The result of the enumeration is a set of patterns $\mathscr{D}$ such that any aliasing pattern $\beta$ is deadlock-causing iff it subsumes a pattern $\alpha \in \mathscr{D}$.

**Lemma 4.2** *An aliasing pattern $\alpha$ is safe iff for all $\beta \in \mathscr{D}$, $\beta \nsubseteq \alpha$.*

*Proof* We prove this by contradiction. Suppose $\alpha$ is safe, and there exists $\beta \in \mathscr{D}$, s.t. $\beta \subseteq \alpha$. By definition, $\alpha$ subsumes $\beta$; hence by Lem. 4.1, if $\beta$ is deadlock-causing, $\alpha$ is deadlock-causing, which is a contradiction. □

In practice, contract derivation consists of first *compacting* the set $\mathscr{D}$ to obtain the minimal deadlock-causing patterns. The contract for safe calling contexts can then be expressed succinctly using the fact that any such pattern must not subsume any element of the set $\mathscr{D}$.

**Example 4.3** From Table 4.1, the only unsafe pattern enumerated is

$$\alpha_2 : \{(\texttt{ob}_1, \texttt{ob}_2\texttt{.nextQueue}), (\texttt{ob}_2, \texttt{ob}_1\texttt{.nextQueue})\}$$

The set of safe patterns therefore is specified by the following set:

$$\left\{\alpha \;\middle|\; (\texttt{ob}_1, \texttt{ob}_2\texttt{.nextQueue}) \notin \alpha \text{ or } (\texttt{ob}_2, \texttt{ob}_1\texttt{.nextQueue}) \notin \alpha \right\}.$$

In terms of a *contract*, this set is expressed as

$$\neg\text{isAliased}(\texttt{ob}_1, \texttt{ob}_2\texttt{.nextQueue}) \vee \neg\text{isAliased}(\texttt{ob}_2, \texttt{ob}_1\texttt{.nextQueue}).$$

**Theorem 4.1** *The set $\mathscr{D}$ of deadlock-causing alias patterns for each pair of library methods obtained by the enumeration technique in Algo. 4 yields a* contract *of the form:*

$$\bigwedge_{\alpha \in \mathscr{D}} \bigvee_{(e_i, e_j) \in \alpha} \neg isAliased(e_i, e_j).$$

Note that the contract is a Boolean combination of propositions conjecturing aliasing between access expressions. Thus, such a contract can be both statically and dynamically enforced in a client, as the concrete aliasing information between access expressions can be obtained through alias analysis, or may be available at run-time.

## 5 Deadlocks in Signaling-based Synchronization

So far we have looked at deadlocks arising from circular dependencies in lock acquisition. Recall that a lock is an abstraction for specifying mutual exclusion, and is implemented as `Java` monitors or `pthread` mutexes. In `Java`, each object monitor is provided with `wait` and `notify` methods to achieve signaling-based synchronization. Recall the semantics of `wait` and `notify` methods from Sec. 2. As before, we assume that the library is intended to be well-encapsulated: every `wait` statement in some library method is expected to have a matching `notify` statement from within some (possibly the same) library method.

*Happens Before.*
We define a *happens before* relation similar to [19] applied to concurrent systems (cf. [13]). In simple terms, a statement $s_1$ happens before $s_2$ (denoted $s_1 \to s_2$) if a causal precedence can be established between the execution of $s_1$ by $T$ and $s_2$ by some (possibly the same) thread $T'$. For instance, statements in the same thread are trivially ordered by $\to$, and causal

precedence is established across statements in different threads by synchronization operations (such as lock release, lock acquisition, thread notification, *etc.*).

A thread executing a method that contains a statement $s = \texttt{mon.wait}()$ is suspended upon executing $s$. It is understood that a waiting thread will be eventually notified by another thread that executes the statement $r = \texttt{mon.notify}()$, failing which the waiting thread fails to progress (causing a deadlock). Such a lack of notification can be ascribed to either a *missing*, or a *lost* or an *unreachable* notification. We elaborate on these scenarios as follows:

I. For some `wait()` statement, there is no matching `notify()` statement in the library. This is an instance of a missing notification.
II. For some concurrent execution, it is possible that every notify statement $r$ satisfies that $r \to s$. This is an instance of a lost notification, *i.e.*, the appropriate object *is* notified, but *before* it has a chance to wait.
III. Assume that for every `wait` statement $s$, there is a matching `notify` statement $r$ present in some library method (*i.e.*, notify is *not* missing). For some concurrent execution, $s \to r$, but $r$ is unreachable in a possible notifying method.

Analyzing programs with wait-notify synchronization is hard. In fact, context-sensitive synchronization-sensitive analysis is undecidable [27]. As the general problem is undecidable, we use a case-by-case analysis to identify sub-problems with conservative solutions.

To statically detect Case I, we can make use of a thread-aware, thread-safe alias analysis: For a given pair of methods (such that one invokes a `wait`, and other a `notify`), we need to check if the `wait` and `notify` are invoked on objects that alias to each other. We remark that this case subsumes the common beginner mistake of `Java` programmers to invoke `wait` and `notify` on the `this` object from within two different monitors. The result is that the `wait` is issued on one monitor, while the `notify` is issued on an entirely different monitor, causing the waiting thread to wait forever.

Case II requires semantic analysis of the code to deduce whether there is an interleaving in which a `notify` precedes the matching `wait` statement. Thus, case II is tricky to detect statically without introducing a slew of false positives. It may be possible to predict such deadlocks by conservative approximation of the happens before relation; however, this is beyond the scope of this paper.

Case III can manifest due to different reasons, such as (a) the `notify` is in a control-flow path that the "notifying" method does not execute, (b) waits and notifies are mismatched, and (c) methods acquire monitors in a nested fashion. As in Case II, (a) requires semantic analysis, and goes beyond the scope of this paper. Examples of Case III(b) include cases where there are more `wait` statements that `notify` statements, and there is a circular dependency between `wait` and `notify` instructions,*i.e.*, $T_1$ executes $\texttt{mon1.wait}()$ followed by $\texttt{mon2.notify}()$, while $T_2$ executes $\texttt{mon2.wait}()$ followed by $\texttt{mon1.notify}()$. We can extend our approach to statically detect these kind of deadlocks by checking compatibility between `wait-notify` sequences as a part of future work. In what follows we focus on Case III(c).

## 5.1 Nested Monitor Deadlocks

Nested monitor deadlocks have been well-known in the literature since before the advent of programming languages that use monitors, cf. [21]. We wish to analyze potential nested monitor deadlocks in concurrent libraries written in languages such as `Java` that use `wait-notify` constructs. As before, we assume that library methods are invoked by separate threads, and use the terms threads and methods interchangeably.

The semantics of `wait-notify` create situations that not only lead to unpleasant deadlocks, but also violate the well-encapsulation principle of modular software. We explain this with an example.

**Example 5.1** The library method `foo` contains a `this.wait()` statement within the monitored region for `this`. `foo` is invoked on the object `lib` from within the monitored region for `mon1` inside the client method `bar`. As the acquisition of `this` (i.e. `lib`) is nested within the scope of `mon1`, this is an instance of a nested monitor acquisition. Recall the semantics of `wait` from Sec. 2. For a thread *T* executing the `this.wait()` statement, the effect is that *T* releases the monitor associated with `this`, but not the monitor `mon1`. Thus, a library method `foo` holds on to a client resource, which violates the spirit of well-encapsulation.

```
1: public class Library {
2:     public synchronized void foo() {
3:         this.wait();
4:     }
5: }
```

```
1: public class Client {
2:     Library lib;
3:     public void bar() {
4:         synchronized (mon1) {
5:             lib.foo();
6:         }
7:     }
8: }
```

Thus, we argue that among all the different cases in which deadlocks manifest in concurrent libraries using `wait-notify`, these kind of deadlocks are the most important to document and predict. We now give concrete examples of nested monitor deadlocks due to `wait-notify`.

*Unreachable Notification.*
Consider the case where a method *m* acquires some monitors before executing `mon.wait()`. As per the semantics of `mon.wait()`, (the thread executing) *m* releases `mon`, but while waiting, it still *holds* the previously acquired monitors. Now, any method $m'$ that needs to acquire one of the "held" monitors before it can execute $s_2$ (`mon.notify()`) will never reach $s_2$. Thus, method *m* waits for some method to notify it, while $m'$ waits for the locks held by *m*[7]. In the literature, such a deadlock has also been called a *hold and wait* deadlock. We discuss two examples of such a deadlock.

**Example 5.2** Consider the code shown below. In Line 4, method `m1_w` releases the monitor `mon2`, but still holds `mon1`. As a result, `m1_n` can never reach Line 4 as it remains "stuck" in the entry set of `mon1` at Line 2. As both `m1_w` and `m1_n` cannot progress, this could lead to a deadlock.

```
1: public void m1_w () {
2:     synchronized (mon1) {
3:         synchronized(mon2) {
4:             mon2.wait();
5:         }
6:     }
7: }
```

```
1: public void m1_n () {
2:     synchronized (mon1) {
3:         synchronized (mon2) {
4:             mon2.notify();
5:         }
6:     }
7: }
```

[7] It is possible that there is a third method $m''$ in which `notify` is reachable, and it can issue a notification. However, while checking possibility for a deadlock between concurrent invocation of *m* and $m'$, we err on the conservative side, and do not make assumptions about the existence of such a $m''$.

**Example 5.3** Methods `m2_w` and `m2_n` shown below have a similar situation as in Ex. 5.2. Method `m2_w` releases `mon1` in Line 4, but still holds `mon2`. Thus, `m2_n` cannot proceed beyond Line 3, where it gets stuck in the entry set of `mon2`.

```
1: public void m2_w () {
2:     synchronized (mon1) {
3:         synchronized(mon2) {
4:             mon1.wait();
5:         }
6:     }
7: }
```

```
1: public void m2_n () {
2:     synchronized (mon1) {
3:         synchronized (mon2) {
4:             mon1.notify();
5:         }
6:     }
7: }
```

*Deadlock due to lock-order inversion.*
In addition to the deadlocks due to unreachable notification, nested monitors also cause deadlocks due to an inversion in the lock acquisition order. We explain this scenario in Ex. 5.4.

**Example 5.4** Consider the following interleaved execution of the methods `m3_w` and `m3_n` shown below:

```
1: public void m3_w {
2:     synchronized (mon1) {
3:         synchronized(mon2) {
4:             mon1.wait();
5:         }
6:     }
7: }
```

```
1: public void m3_n {
2:     synchronized (mon1) {
3:         mon1.notify();
4:         synchronized(mon2) {
5:                 ...
6:         }
7:     }
8: }
```

Method `m3_w` holds the monitor `mon2`, and is in the wait set of `mon1` at Line 4. Method `m3_n` acquires `mon1`, and then calls `mon1.notify()` (Line 3). Upon waking up, `m3_w` first tries to re-acquire `mon1`. However, as `m3_n` holds `mon1`, `m3_w` cannot proceed. On the other hand, `m3_n` tries to acquire `mon2`, but as `m3_w` holds `mon2`, `m3_n` cannot progress beyond Line 4. Effectively, due to the semantics of `wait`, the lock-acquisition order gets reversed, causing the classic cyclic dependency deadlock that we discussed in the previous sections.

## 6 Generalized Nested Monitor Rule

In this section, we show how we can extend the lock-graph computation in Sec. 3.1 to capture deadlocks induced by nested monitors.

### 6.1 Extended Lock-Graph

We extend the notion of a lock-acquisition order graph defined in Sec. 3.1. Consider the set of control-flow edges shown in Fig. 6.1. We now formulate static rules that help in construction of an *extended lock-graph* that models the nested monitor deadlocks resulting from unreachable notification and lock-order inversion. We recall the rule for modeling the edge $e_l$ in Rule R0 from Sec. 3.1. Recall that the set of locks held by $m$ before executing a statement $s$ is $\texttt{lockset}(u)$ for the edge $u \xrightarrow{s} v$ in $cfg(m)$. We denote the extended lock-graph by $elg(V,E)$. Rules R1, R2 specify how to model the edge $e_w$ for a `wait` statement, while Rule R3 specifies how to model the edge $e_n$ for a `notify` statement.

$$\forall \ell \in \texttt{lockset}(u_0), \quad \text{add edge} \ (\ell, \texttt{mon}) \tag{R0}$$

$$\forall \ell \in \texttt{lockset}(u_1),\ \ell \neq \texttt{mon}, \quad \text{add edge} \ (\ell, \texttt{smon}) \tag{R1}$$

$$\forall \ell \in \texttt{lockset}(u_1),\ \ell \neq \texttt{mon}, \quad \text{add edge} \ (\ell, \texttt{mon}) \tag{R2}$$

$$\forall \ell \in \texttt{lockset}(u_2),\ \ell \neq \texttt{mon}, \quad \text{add edge} \ (\texttt{smon}, \ell) \tag{R3}$$

**Definition 6.1 (Distinct Cycle)** A cycle in the merged extended lock-graph is called a distinct cycle if each edge in the cycle is induced by a distinct method (invocation).

**Lemma 6.1 (Generalized Nested Monitor Rule)**
*Let the extended lock-graphs obtained using Rules R1, R2, R3 for methods $m_1, \ldots, m_k$ be $elg(m_1), \ldots, elg(m_k)$. Concurrent calls to methods $m_1, \ldots, m_k$ may deadlock if $\bigsqcup_{i=1}^{k} elg(m_i)$ contains a distinct cycle.*

*Proof* Recall that in Rules R1, R2, R3, we only focus on deadlocks due to nested monitor acquisition leading to unreachable notification or lock-order inversion. We argue that in both scenarios, whenever methods $\mathtt{m_w}$ and $\mathtt{m_n}$ deadlock, the merged extended lock-graph $elg(m_w) \sqcup elg(m_n)$ contains a cycle. Thus, the presence of a cycle indicates the possibility of a deadlock.

Consider the control-flow edges shown in Fig. 6.1. An unreachable notification occurs when $\mathtt{m_w}$ holds some lock `mon` and waits, while method $\mathtt{m_n}$ needs lock `mon` before it can reach the notification that "wakes up" $\mathtt{m_w}$. For a method $m_w$ executing $s_1$, Rule R1 mimics the act of relinquishing `mon` and waiting. As $\mathtt{m_w}$ holds all the locks in $\texttt{lockset}(u_1)$ (except for `mon`) when waiting, we create a vertex `smon` in the lock-graph, and add edges from every monitor $\ell$ (except `mon`) in $\texttt{lockset}(u_1)$ to `smon`. Method $\mathtt{m_n}$ (containing statement $s_2$) can reach $s_2$ if it can successfully acquire all the monitors in in $\texttt{lockset}(u_2)$ (except `mon`). We model this by adding edges from `smon` to each monitor in $\texttt{lockset}(u_2)$ (Rule R3). Thus, rules R1 and R3 ensure that if monitor $\ell$ is held by $m_w$ when it starts waiting, and $m_n$ needs to acquire $\ell$ to reach the notification, then $elg(m) \sqcup elg(m')$ contains a cycle of the form: $\ell \xrightarrow{R1,m_w} \texttt{smon} \xrightarrow{R3,m_n} \ell$. Note that since the edges involved in this cycle are necessarily from different methods, such a cycle is a distinct cycle.

Rule R2 encodes lock-order inversion. Suppose $m_w$ holds certain monitors, and is waiting as a result of a call to `mon.wait()`. Upon waking up, $m_w$ tries to re-acquire `mon`; so we add dependency edges $(\texttt{mon}', \texttt{mon})$, where $\texttt{mon}'$ is some monitor (other than `mon`) held by $m_w$ before executing `mon.wait()`. If *another* method, say $m_n$ requires acquisition of `mon` before

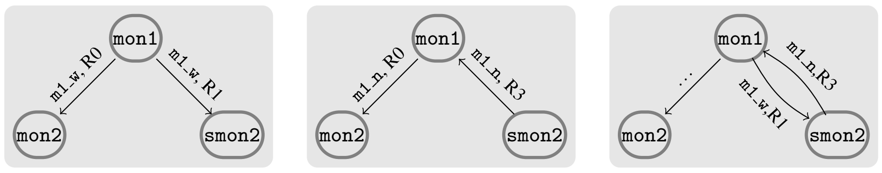
$$\begin{aligned}
e_l &: \ (u_0 \xrightarrow{s_0\,=\,\texttt{lock(mon)}} v_0) \\
e_w &: \ (u_1 \xrightarrow{s_1\,=\,\texttt{mon.wait()}} v_1) \\
e_n &: \ (u_2 \xrightarrow{s_2\,=\,\texttt{mon.notify()}} v_2)
\end{aligned}$$

Fig. 6.1: Control-flow Edges

acquiring `mon`$'$, then the concurrent invocation of $m_n$ and $m_w$ could deadlock. This scenario corresponds to a distinct cycle of the form: $\texttt{mon}' \xrightarrow{R2,m_w} \texttt{mon} \xrightarrow{R0,m_n} \texttt{mon}'$.

Lastly, above arguments can be inductively extended to the case where there are multiple waiting and notifying methods. □

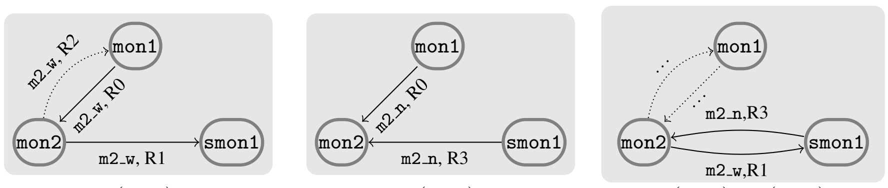
**Example 6.1** Consider the extended lock-graphs for `m1_w` and `m1_n` from Ex. 5.2:



$lg(\texttt{m1\_w})$ $lg(\texttt{m1\_n})$ $lg(\texttt{m1\_w}) \sqcup lg(\texttt{m1\_n})$

We can observe that there is a distinct cycle in lg$(\texttt{m1_w}) \sqcup lg(\texttt{m1_n})$.

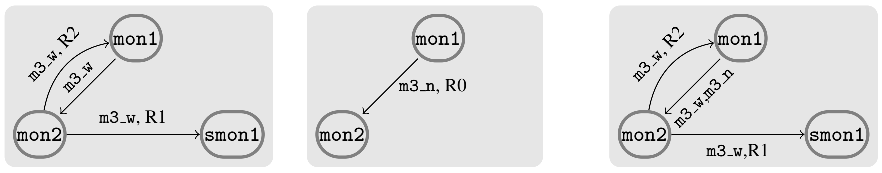
**Example 6.2** Consider the extended lock-graphs for `m2_w` and `m2_n` from Ex. 5.3:



$lg(\texttt{m2\_w})$ $lg(\texttt{m2\_n})$ $lg(\texttt{m2\_w}) \sqcup lg(\texttt{m2\_n})$

In this case, there are two distinct cycles in the merged graph indicating potential deadlocks. The cycle due to unreachable notification corresponds to the solid edges.

**Example 6.3** Consider the extended lock-graphs for `m3_w` and `m3_n` from Ex. 5.4:



$lg(\texttt{m3\_w})$ $lg(\texttt{m3\_n})$ $lg(\texttt{m3\_w}) \sqcup lg(\texttt{m3\_n})$

There is a distinct cycle between nodes `mon1` and `mon2` by picking the label `m3_n` for the edge (`mon1`,`mon2`) and `m3_w` for (`mon2`,`mon1`), which indicates a potential deadlock.

## 6.2 Modifications to Lock-Graph Computation

The algorithm for lock-graph computation outlined in Sec. 3.1 can be extended to accommodate the generalized nested monitor rule. Recall that the summary computed for each point in the control-flow graph of a method $m$ is a tuple $(lg(V,E), \texttt{lockset}, \texttt{roots})$ where $lg(V,E)$ is the lock-order graph, `lockset` is the set of locks currently acquired by method $m$, and `roots` is the set of locks that do not have parent nodes in $lg$. For computing the extended lock-graphs, the partial summary is defined as the tuple: $(elg(V,E), \texttt{lockset}, \texttt{roots}, \texttt{notifySet})$,

**Function** `extendedComputeFlow`

```
input  : cfg edge: $u \xrightarrow{s} v$, Method name: $m$
output: partial summary $out$
1  begin
2      $in := \mathtt{psum}(u)$, $out := \emptyset$
       /* $in$ = $(\mathtt{V}_i,\mathtt{E}_i,\mathtt{roots}_i,\mathtt{lockset}_i,\mathtt{notifySet}_i)$, $out$ =
          $(\mathtt{V}_o,\mathtt{E}_o,\mathtt{roots}_o,\mathtt{lockset}_o,\mathtt{notifySet}_o)$ */
3      switch $(s)$ do
4          Lines 4-16 of function computeFlow
           /* Wait Statement. */
5          case mon.wait() ():
6              foreach $(\mathtt{mon}' \in \mathtt{lockset}_i)$ s.t. $(\mathtt{mon}' \neq \mathtt{mon})$ do
                   /* Rule R1: */
7                  $\mathtt{E}_o := \mathtt{E}_o \cup \{(\mathtt{mon}' \xrightarrow{m} \mathtt{smon})\}$
                   /* Rule R2: */
8                  $\mathtt{E}_o := \mathtt{E}_o \cup \{(\mathtt{mon}' \xrightarrow{m} \mathtt{mon})\}$
           /* Notify Statement. */
9          case mon.notify() ():
10             foreach $(\mathtt{mon}' \in ls_i)$ s.t. $(\mathtt{mon}' \neq \mathtt{mon})$ do
                   /* Rule R3: */
11                 $\mathtt{E}_o := \mathtt{E}_o \cup \{(\mathtt{smon} \xrightarrow{m} \mathtt{mon}')\}$
12                 $\mathtt{notifySet} := \mathtt{notifySet} \cup \{\mathtt{smon}\}$
           /* Method Invocation. */
13         case $\mathtt{m}'(a_1,\ldots,a_k)$ :
14             Lines 12-16 of function computeFlow
               /* Recall, $sum'$ = $\mathtt{summary}(m')|_{\forall i: f_i \mapsto a_i}$ */
               /* Also, $sum' = (\mathtt{V}',\mathtt{E}',\mathtt{lockset}',\mathtt{roots}',\mathtt{notifySet}')$ */
15             foreach $\mathtt{smon} \in \mathtt{notifySet}'$ do
16                 foreach $\mathtt{mon} \in \mathtt{lockset}$ do
17                     $\mathtt{E}_o := \mathtt{E}_o \cup \{(\mathtt{smon} \xrightarrow{m} \mathtt{mon})\}$
18             $\mathtt{notifySet} := \mathtt{notifySet} \cup \mathtt{notifySet}'$
19 end
```

where *elg* is the extended lock-graph as defined in Sec. 6.1, and `notifySet` is the set of monitors on which `mon.notify()` has been invoked.

The extendedComputeFlow function specifies the flow equations for the `wait-notify` statements, and modified flow equations for method calls. For a given method $m_1$, the edges induced by Rules R1 and R2 are fully contained within $\mathtt{summary}(m_1)$, and are added in standard fashion (Lines 7, 8). However, the edges induced by Rule R3 are "reverse" edges that can point to nodes outside of $\mathtt{summary}(m_1)$. Consider the case where $m$ calls $m_1$ and $m_1$ contains a `mon.notify()` statement. Now, by Rule R3, we add edges from `mon` (which is a node in $elg(m_1)$) to every lock in `lockset` at the call-site of $m_1$ (which is in $m$). To ease this computation, we add `mon` to the set `notifySet`, when `mon.notify()` is called (Line 12). Let $u \xrightarrow{s=m_1(a_1,\ldots,a_k)} v$ be the call-site of method $m_1$ in $cfg(m)$. When we concatenate $\mathtt{summary}(m_1)$ with the partial summary at point $u$ in $m$ (*i.e.* $\mathtt{psum}(u)$), we add edges from the set `notifySet` in $\mathtt{summary}(m_1)$ to every monitor in $\mathtt{lockset}(u)$ in $\mathtt{psum}(u)$ (Line 15-17).

Recall that for a given pair of methods $m_1$ and $m_2$, there is a potential deadlock if $elg(m_1) \sqcup elg(m_2)$ contains a *distinct cycle*. For deadlockability analysis, we can extend the above rule as follows: $\alpha$ is a deadlock-causing aliasing pattern for methods $m_1$ and $m_2$ if $\alpha \triangleright elg(m_1) \sqcup elg(m_2)$ contains a distinct cycle. We can extend the symbolic encoding described in Sec. 3.3 to encode the extended lock-graphs. The main difficulty is that an extended lock-graph $elg(m)$ for a method $m$ can have cycles, while our symbolic encoding for graphs and aliasing patterns relies on encoding *acyclic* graphs.

We observe that a cycle is added to $elg(m)$ due to simultaneous application of Rule R0 and R2. For instance, consider the case where $m$ acquires monitor `mon1` followed by `mon2`, and then executes `mon1.wait()`. Rule R0 requires an edge to be added from `mon1` to `mon2`, while Rule R2 requires an edge to be added from `mon2` to `mon1`. To check for existence of a distinct cycle, we need either the edge $(\mathtt{mon1}, \mathtt{mon2})$, *or*, the edge $(\mathtt{mon2}, \mathtt{mon1})$, but not both. In effect, we can decompose $elg(m)$ into two acyclic graphs $elg_1(m)$ and $elg_2(m)$, each of which contains exactly one of these two edges. It is easy to see that such a disjunctive decomposition of the extended lock-graph can be systematically performed by "breaking cycles" formed by the symmetric edges induced by R0 and R2.

**Example 6.4** Consider the method `m2_w` shown in Ex. 5.3, and its corresponding extended lock-graph shown in Ex. 6.2. The encoding for $\mathrm{lg}(\mathtt{m2\_w})$ is expressed as two conjunctions as follows:

$$\Psi(G_1) = \underbrace{x(\mathtt{mon1}) < x(\mathtt{mon2})}_{R0} \;\wedge\; x(\mathtt{mon2}) < x(\mathtt{smon1})$$
$$\Psi(G_2) = \underbrace{x(\mathtt{mon2}) < x(\mathtt{mon1})}_{R2} \;\wedge\; x(\mathtt{mon2}) < x(\mathtt{smon1})$$

**Example 6.5** Continuing with Ex. 6.2 and Ex. 6.4, let $\alpha$ be the empty aliasing pattern. Consider the merged lock-graphs obtained by merging $G_1$ and $G_2$ individually with the lock-graph for `m2_n`.

$$\Psi(\alpha, G_1 \sqcup lg(\mathtt{m2\_n})) = \begin{bmatrix} x(\mathtt{mon1}) < x(\mathtt{mon2}) \;\wedge\; x(\mathtt{mon2}) < x(\mathtt{smon1}) \\ x(\mathtt{mon1}) < x(\mathtt{mon2}) \;\wedge\; x(\mathtt{smon1}) < x(\mathtt{mon2}) \end{bmatrix}$$
$$\Psi(\alpha, G_2 \sqcup lg(\mathtt{m2\_n})) = \begin{bmatrix} x(\mathtt{mon2}) < x(\mathtt{mon1}) \;\wedge\; x(\mathtt{mon2}) < x(\mathtt{smon1}) \\ x(\mathtt{mon1}) < x(\mathtt{mon2}) \;\wedge\; x(\mathtt{smon1}) < x(\mathtt{mon2}) \end{bmatrix}$$

We can see that both conjunctions are unsatisfiable. Moreover, each conjunction encodes a distinct cycle in the merged extended lock-graph.

**Example 6.6** Consider Ex. 6.3. Let $\alpha$ be the empty aliasing pattern. Note that $lg(\mathtt{m3\_w})$ in Ex. 6.3 is identical to $lg(\mathtt{m2\_w})$ in Ex. 6.2, and thus the decomposition of $elg(\mathtt{m3\_w})$ into graphs $G_1$ and $G_2$ is as in Ex. 6.4. Now consider the merged lock-graphs obtained by merging $G_1$ and $G_2$ with $lg(\mathtt{m3\_n})$ from Ex. 6.3.

$$\Psi(\alpha, G_1 \sqcup lg(\mathtt{m3\_n})) = \begin{bmatrix} x(\mathtt{mon1}) < x(\mathtt{mon2}) \;\wedge\; x(\mathtt{mon2}) < x(\mathtt{smon1}) \;\wedge \\ x(\mathtt{mon1}) < x(\mathtt{mon2}) \end{bmatrix}$$
$$\Psi(\alpha, G_2 \sqcup lg(\mathtt{m3\_n})) = \begin{bmatrix} x(\mathtt{mon2}) < x(\mathtt{mon1}) \;\wedge\; x(\mathtt{mon2}) < x(\mathtt{smon1}) \;\wedge \\ x(\mathtt{mon1}) < x(\mathtt{mon2}) \end{bmatrix}$$

We can see that the second conjunction is unsatisfiable, and corresponds to a distinct cycle in the merged extended lock-graph for $t_1$ and $t_2$.

Thus, we can observe that if the extended lock-graph $elg(m_1)$ for a method $m_1$ has cycles, then we can decompose it into components $\{elg_1(m_1) \dots elg_n(m_1)\}$, s.t. each $elg_i(m_1)$ is acyclic and $elg(m_1)$ is the union of the components, *i.e.*, $elg(m_1) = \bigcup_{i=1}^{n} elg_i(m_1)$. We can see from the above examples that for a pair of methods $m_1$, $m_2$, the merged extended lock-graph $elg(m_1) \sqcup elg(m_2)$ has a distinct cycle if there exist some acyclic components $elg_i(m_1)$ and $elg_j(m_2)$ such that $elg_i(m_1) \sqcup elg_j(m_2)$ has a cycle. This is formalized in the theorem below.

**Theorem 6.1** *Let $G_i$ be some acyclic component of $elg(m_1)$ and $G_j$ be some acyclic component of $elg(m_2)$. Let $G_{ij}$ denote $G_i \sqcup G_j$. The formula $[\Psi(\alpha, G_{ij})]$ is satisfiable for all $i, j$ iff $[\alpha \triangleright (elg(m_1) \sqcup elg(m_2))]$ does not have a distinct cycle.*

*Proof* We give a proof outline:

1. We first prove that $[\alpha \triangleright elg(m_1) \sqcup elg(m_2)]$ does not contain a distinct cycle iff $\forall i$ and $\forall j$, $elg_i(m_1) \sqcup elg_j(m_2)$ does not contain a cycle. This follows from the definition of the decomposition operation.
2. If $elg_i(m_1) \sqcup elg_j(m_2)$ does not contain a cycle, then by Theorem 3.1, we know that $\Psi(\alpha, G_{ij})$ is satisfiable. Thus if $\forall i$ and $\forall j$, if $elg_i(m_1) \sqcup elg_j(m_2)$ does not contain a cycle, then $\forall i$ and $\forall j$, $\Psi(\alpha, G_{ij})$ is satisfiable. $\square$

## 7 Analyzing Clients

The interface contracts generated by our tool vastly simplify the analysis of client code that makes use of the library methods that are part of the library's interface contract. Furthermore, they serve to document against the improper use of the methods in a multi-threaded context.

We recall from Section 4 that the final contract for a safe call to a pair of methods $m_1, m_2$ is a Boolean expression involving propositions of the form $\neg$isAliased($\mathtt{e}_i, \mathtt{e}_j$), wherein $\mathtt{e}_i$ and $\mathtt{e}_j$ are access expressions corresponding to the formal parameters of the methods, including the "`this`" parameter.

In practice, checking such a contract for a given client that uses the library involves two major components: (A) a *May-happen in Parallel* (MHP) analysis [20] for calls to methods $m_1$ and $m_2$ to determine if two different threads may reach these method call-sites simultaneously, and (B) a conservative, thread-safe alias analysis in order to determine the potential aliasing of parameters at the invocation sites of the methods in question.

On the basis of such an alias analysis, we may statically evaluate the contract at each concurrent call-site. Note that these two components are already part of most data-race detection tools such as *CHORD* [23, 24]. In theory, deadlock violations can be directly analyzed by a "whole-program analysis" of the combined client and the library code. In practice, this requires the (re-)analysis of a significant volume of code. Using contracts has the distinct advantage of being fast in the case of small clients that invoke a large number of library methods. Moreover, decoupling the client analysis from the library analysis allows our technique to be compositional. Since library internals are often confusing and opaque to the client developers, another key advantage is the ability to better localize failures to their causes in the clients, as opposed to causes inside the library code.

Table 8.1: Experimental Results

| Library | KLOC | Num. of Alias | Num. of DL- | Computation Time (secs)[a] | | Num. of Unique Scenarios | |
|---|---|---|---|---|---|---|---|
| | | Patterns Checked | causing Patterns | Lock-Graph | SMT Solver | False Positives | Potential Deadlocks |
| apache-log4j | 33.3 | 4 | 4 | 130 | 0.1 | 1 | 1 |
| cache4j | 2.6 | 0 | 0 | 15 | - | - | - |
| ftpproxy | 1.0 | 0 | 0 | 13 | - | - | - |
| hsqldb | 157.6 | 369 | 231 | 804 | 2.8 | 3 | 3 |
| JavaFTP | 2.6 | 0 | 0 | 9 | - | - | - |
| netty | 11.0 | 0 | 0 | 14 | - | - | - |
| oddjob | 41.3 | 0 | 0 | 250 | - | - | - |
| java.applet | 0.9 | 102 | 64 | 64 | 1.0 | 1 | 1 |
| java.awt | 163.9 | 5325 | 3800 | 454 | 26.4 | 2 | 3 |
| java.beans | 16.2 | 148 | 108 | 31 | 1.5 | 1 | 2 |
| java.io | 28.6 | 32 | 0 | 39 | 0.0 | - | - |
| java.lang | 55.0 | 279 | 89 | 46 | 1.9 | 3 | 2 |
| java.math | 9.1 | 0 | 0 | 18 | - | - | - |
| java.net | 26.5 | 55 | 44 | 32 | 0.5 | 1 | 1 |
| java.nio | 46.7 | 0 | 0 | 19 | - | - | - |
| java.rmi | 9.1 | 2 | 2 | 14 | 0.1 | 1 | 0 |
| java.security | 34.2 | 0 | 0 | 27 | - | - | - |
| java.sql | 22.2 | 1836 | 0 | 10 | 8.0 | - | - |
| java.text | 22.6 | 26 | 18 | 26 | 0.2 | 1 | 0 |
| java.util | 116.8 | 188 | 117 | 190 | 2.0 | 4 | 3 |
| javax.imageio | 24.7 | 0 | 0 | 22 | - | - | - |
| javax.lang | 5.2 | 0 | 0 | 8 | - | - | - |
| javax.management | 67.5 | 16 | 6 | 74 | 0.2 | 2 | 0 |
| javax.naming | 19.5 | 0 | 0 | 64 | - | - | - |
| javax.print | 2.1 | 2 | 0 | 27 | - | - | - |
| javax.security | 11.7 | 164 | 110 | 27 | 1.2 | 2 | 0 |
| javax.sound | 14.3 | 0 | 0 | 10 | - | - | - |
| javax.sql | 18.2 | 0 | 0 | 14 | - | - | - |
| javax.swing | 322.2 | 132 | 120 | 353 | 1.6 | 2 | 2 |
| javax.xml | 48.9 | 0 | 0 | 27 | - | - | - |

[a] All experiments were performed on a Linux machine with an AMD Athlon 64x2 2.2 GHz processor, and 6GB RAM.

## 8 Experimental Results

We have implemented a prototype tool for synthesizing interface contracts for `Java` libraries. The tool consists of a summary based lock-order graph analysis for a given `Java` library followed by its encoding into logical formulae for symbolic enumeration of alias patterns. We utilize the `soot` framework for implementing the lock-order graph extraction [29], using custom analyses for alias propagation built atop `soot`'s native intraprocedural alias analysis. Before generating constraints for analysis with the SMT solver, we prune the lock-order graphs using various filtering strategies strategies (in addition to those discussed in Sec. 4):

(a) Pruning *unaliasable* fields, *e.g.* `final` fields initialized to a constant.
(b) Removing objects declared `private` that are not accessed outside the constructor or finalizer.
(c) Removing immutable string constants and `java.lang.Class` constants.

(d) Pruning objects that cannot escape the scope of a given library method using a conservative *escape analysis*.

These filtering strategies discussed above are sound: our tool does not miss any potential deadlock due to the above strategies. For example, we prune an object from the lock-order graph only if it definitely cannot escape the scope of a library method, or is aliased to an immutable constant. In other words, we only remove a node from the lock-graph if it is impossible for it to be aliased to any other node, and thus impossible to be part of some cycle.

The generated constraints are solved using the SMT solver Yices [11]. Table 8.1 summarizes the potential deadlocks thus obtained. Table 8.1 shows that our tool runs in a relatively short amount of time even for large `Java` libraries. Furthermore, the runtime is dominated by the lock-order graph computation rather than the enumeration and constraint solving with the SMT solver.

Some deadlock-causing aliasing patterns are *false positives*. These patterns result from two main sources: a) the static lock-order graph construction is a *may* analysis, and hence there are inaccurate edges and nodes in the lock-order graph, and b) our alias analysis is a *may* analysis, which leads to aliasing patterns that cannot be realized. We manually examine the output of our tool to discard such patterns. However, the output of our tool may also consist of a large number of "redundant" deadlock-causing patterns. These patterns that are repeated instantiations of the same underlying deadlock scenario, and appear due to the fact that several library methods typically invoke the same deadlock-prone utility method. Such a deadlock gets reported multiple times in our current implementation, each under a different set of library entry methods. The table shows the number of unique scenarios after considering such redundancies (manually, at present).

**Example 8.1** From the lock-order graph of `postEventPrivate` presented in Sec. 1.1, if we concurrently invoke the method `postEventPrivate` on two separate objects `a` and `b`, then it leads to a deadlock under a specific aliasing pattern. However, the methods `postEvent`, `push` and `pop` in the same class also invoke the `postEventPrivate` method, and hence are susceptible to the *same* deadlock. Thus, for each pair of these methods, the same underlying deadlock-causing aliasing pattern is generated. In our experiments, we observed 324 possible deadlock-causing aliasing patterns, all of which correspond to this single unique scenario involving calls to `postEventPrivate`.

Significantly, our tool predicts deadlocks that are highly relevant to some of the clients using these libraries. Some have already manifested in real client code, and have been reported as bugs by developers in various bug repositories. Table 8.2 summarizes the library name and the bug report locations we have found using a web search. Inspection of the bug reports reveals that the aliasing patterns at the call-sites of the methods involved in the deadlock, correspond to a violation of the interface contract for that library, as generated by our tool. Further examples of such deadlocks can be expected in the future. Finally, we remark that the tool currently analyzes only deadlocks arising from circular dependencies in lock acquistion, and extensions to the tool to incorporate the heuristics for detecting deadlocks in `wait-notify` programs will be an important part of the future work.

*Sources of Unsoundness.* Our strategies for pruning lock-order graphs outlined earlier in this section are sound: no deadlocks can be missed as a result of removing nodes from the lock-order graphs under consideration. However, to ensure that our analysis terminates, we use certain abstractions that could lead to unsoundness. For instance, we bound the size

| Library Name | Method names | Bug Report |
|---|---|---|
| java.awt (EventQueue) | postEventPrivate, wakeup | [9]:4913324 [9]:6424157, [9]:6542185 |
| java.awt (Container) | removeAll, addPropertyChangeListener | [4] |
| java.util (LogManager) (Logger) | addLogger getLogger | [9]:6487638 |
| javax.swing (JComponent) | setFont paintChildren | Jajuk [17] |
| hsqldb (Session) | isAutoCommit close | [3] |

Table 8.2: Real Client Deadlocks

of access expressions when iteratively computing the lock-order graph for recursive methods, which could lead to potentially missing deadlocks. Our analysis ignores the dynamics of thread creation from within a library method, and could miss a potential deadlock if a newly spawned thread contains a circular locking dependency with the parent thread. In case of programs with `wait/notify`, we do not make any guarantees of soundness, as our analysis currently does not handle missing notifications, lost notifications, and unreachable notifications that are not a result of nested monitor invocations. Thus, our contributions for `wait/notify` deadlocks should not be viewed as much more than effective heuristics. Lastly, our technique can only detect deadlocks that manifest as a result of circular dependencies in lock acquisition. Thus, deadlocks that may result from conditions such as insufficient memory, thread joins, interrupts, non-terminating, or any other reason are not detected by our technique.

## 9 Related Work and Conclusions

### 9.1 Related Work

*Runtime Techniques.*
Runtime techniques for deadlock detection track nested lock acquisition patterns. The *Good-Lock* algorithm [14] is capable of detecting deadlocks arising from two concurrent threads; [2] generalizes this to an arbitrary number of threads, and defines a special type system in which potential deadlocks correspond to code fragments that are untypable. Agarwal et al. [1] further extend this approach to programs with semaphores and condition variables.

*Model Checking.*
Model checking techniques [7] have been successfully used to detect deadlocks in programs. For instance, Corbett et al. employ model checking to analyze protocols written in Ada for deadlocks [8]. Model checkers such as *SPIN* [16], *Java Path Finder* [14, 15] have been used extensively to check concurrent `Java` programs for deadlocks. However, program size and complexity limit these approaches in presence of arbitrary aliasing. A compositional technique based on summarizing large libraries can help these approaches immensely. Bensalem

et al. [6] propose a dynamic analysis approach, based on checking synchronization traces for cycles, with special emphasis on avoiding certain kinds of guarded cycles that do not correspond to a realizable deadlock.

*Static Techniques.*
Static techniques based on dataflow analysis either use dataflow rules to compute lock-order graphs [12, 26] or examine well-known code patterns [5, 25] to detect deadlocks. Naik et al. present an interesting combination of different kinds of static analyses to approximate six necessary conditions for deadlock [24]. Most static techniques focus on identifying deadlocks within a given closed program, while in [24], the authors close a given open program (the library) by manually constructing a harness for that program. In [28] the author analyzes the entire `Java` library, and uses a coarser level of granularity in lock-order graph construction.

*Deadlock Detection for Libraries.*
As mentioned previously, deadlock analysis for concurrent libraries was first introduced by Williams et al. [30] for analyzing `Java` libraries. Therein, the authors use types to approximate the may-alias relation across nodes in the lock-order graphs for a library, and reduce checking existence of potential deadlocks to cycle detection. Our approach is inspired by this work and seeks to solve the very same problem under similar assumptions. Our distinct contributions lie in the use of aliasing information in the library. As Williams et al. rightly point out, there is an overwhelming amount of aliasing possible. Therefore, we use pruning as well as symbolic encoding of the aliasing patterns. Our use of subsumption ensures that a tiny fraction of the exponentially many alias patterns are actually explored, and doing so clearly reduces the number of false positives without the use of unsound filtering heuristics. The use of aliasing pattern subsumption also ensures that the final deadlock patterns can be inverted to yield statically enforceable interface contracts.

We introduced symbolic techniques for deadlockability analysis in [10]. In this paper, we extend this methodology to reason about deadlocks in libraries that employ signaling-based synchronization primitives such as `wait-notify` in `Java`. We formulate a generalized nested monitor rule to identify code patterns that can lead to a deadlock, and provide static techniques to detect them. Finally, we enable symbolic reasoning using SMT solvers for this more general case, and present the required encoding and algorithms.

## 9.2 Conclusions and Future Work

The techniques presented thus far identify patterns of aliasing between the parameters of concurrent library methods that may lead to a deadlock. We use these patterns to synthesize interface contracts on the library methods, which can then be either used by developers when writing the client code, or by analysis tools to automate deadlock detection in the client.

Synchronization primitives such as locks and monitors used for enforcing mutual exclusion are the most common source of deadlocks. Hence, the main thrust of this paper is on detecting deadlocks based on cyclic dependencies in the acquisitions of such (lock and monitor) variables. Likewise, in our current implementation, our tool is limited to predicting such deadlocks. Moreover, as we have focused on the analysis of `Java` programs, we assume that the monitors are re-entrant. In this paper, we present a technique to analyze deadlocks in libraries that use signaling-based synchronization with the help of `wait-notify` statements. Validating this extension with experimental results remains an important part of the future work. We would also like to incorporate reasoning about libraries that use (counting) semaphores, and locks with arbitrary re-entrancy models in the future.

For control-flow graphs of libraries that use recursive types, we artificially bound the size of the resulting access expressions, which may lead to a deadlock being missed when analyzing a scenario involving multiple concurrent threads (where the number of threads exceeds this artificial bound on the size). However, since deadlocks involving more than three threads are extremely rare in practice, such artificial bounds do not impact the effectiveness of our tool in identifying real deadlocks.

While the number of false positives generated by our tool is low, cases such as guarded cycles, *i.e.*, cycles that are infeasible as each entry node in the cycle is protected by a common lock [6], are not currently handled. Dealing with newer features of the `Java` language such as *generics*, and `Java`'s concurrency library (`java.util.concurrent`) that uses constructs similar to the `pthread` library is a challenge. The automatic identification of unique scenarios from the interface contracts generated by our current implementation, as well as the static analysis that checks/enforces the derived interface contracts on real client code (as described in Sec. 7), will be completed as part of future work.

**Acknowledgements** We would like to thank Aarti Gupta, Vineet Kahlon, and Franjo Ivančić for insightful discussions during the initial part of this work. We would also like to thank the anonymous reviewers for their helpful comments and suggestions.

## References

1. Agarwal R, Stoller SD (2006) Run-time Detection of Potential Deadlocks for Programs with Locks, Semaphores, and Condition Variables. In: Proc. of Workshop on Parallel and Distributed Systems: Testing and Debugging, pp 51–60
2. Agarwal R, Wang L, Stoller S (2006) Detecting Potential Deadlocks with Static Analysis and Run-Time Monitoring. Hardware and Software, Verification and Testing pp 191–207
3. Open Source Mail Archive (2004) Message #150. URL `http://osdir.com/ml/java.hsqldb.user/2004-03/msg00150.html`
4. Open Source Mail Archive (2008) Bug 159. URL `http://osdir.com/ml/java.openjdk.distro-packaging.devel/2008-06/msg00061.html`
5. Artho C, Biere A (2001) Applying Static Analysis to Large-Scale, Multi-Threaded Java Programs. In: Proc. of the 13th Australian Conference on Software Engineering, p 68
6. Bensalem S, Havelund K (2005) Dynamic Deadlock Analysis of Multi-Threaded Programs. In: Proc. of the Haifa Verification Conference, pp 208–223
7. Clarke EM, Emerson EA (1981) Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In: Logics of Programs, pp 52–71
8. Corbett JC (1996) Evaluating Deadlock Detection Methods for concurrent software. IEEE Transactions on Software Engineering 22(3):161–180
9. Sun Developer Network Bug Database (2007) URL `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=xxxx`, bug-id provided at citation
10. Deshmukh JV, Emerson EA, Sankaranarayanan S (2009) Symbolic Deadlock Analysis in Concurrent Libraries and Their Clients. In: Proc. of the 24th IEEE/ACM International Conference on Automated Software Engineering, pp 480–491
11. Dutertre B, de Moura L (2006) A Fast Linear-Arithmetic Solver for DPLL(T). In: Proc. of Computer Aided Verification, pp 81–94

12. Engler D, Ashcraft K (2003) Racerx: Effective, Static Detection of Race Conditions and Deadlocks. ACM SIGOPS Operating System Review 37(5):237–252
13. Flanagan C, Freund SN, Yi J (2008) Velodrome: a Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. In: Proc. of PLDI, pp 293–303
14. Havelund K (2000) Using Runtime Analysis to Guide Model Checking of Java Programs. In: Proc. of SPIN Workshop on Model Checking of Software, pp 245–264
15. Havelund K, Pressburger T (2000) Model Checking JAVA Programs using Java PathFinder. International Journal on Software Tools for Technology Transfer 2(4):366–381
16. Holzmann GJ (2003) The SPIN Model Checker. Addison-Wesley
17. Jajuk Advanced Jukebox (2008) Bug Ticket #850. URL `http://trac.jajuk.info/ticket/850`
18. Lahiri SK, Musuvathi M (2005) An Efficient Decision Procedure for UTVPI Constraints. In: Proc. of Frontiers of Combining Systems, 5th International Workshop, pp 168–183
19. Lamport L (1978) Time, Clocks, and the Ordering of Events in a Distributed System. Commun ACM 21(7):558–565, URL `http://portal.acm.org.ezproxy.lib.utexas.edu/citation.cfm?id=359563`
20. Li L, Verbrugge C (2004) A Practical MHP Information Analysis for Concurrent Java Programs. In: Proc. of the 17th International Workshop on Languages and Compilers for Parallel Computing, pp 194–208
21. Lister A (1977) The Problem of Nested Monitor Calls. SIGOPS Oper Syst Rev 11(3):5–7
22. de Moura L, Bj$\phi$rner N (2008) Z3: An Efficient SMT Solver. In: Proc. of Tools and Algorithms for the Construction and Analysis of Systems, pp 337–340
23. Naik M, Aiken A, Whaley J (2006) Effective Static Race Detection for Java. In: Proc. of the 2006 ACM SIGPLAN conf. on Programming Language Design and Implementation, ACM, pp 308–319
24. Naik M, Park CS, Sen K, Gay D (2009) Effective Static Deadlock Detection. In: Proc. of the 31st International Conference on Software Engineering, pp 386–396
25. Otto F, Moschny T (2008) Finding Synchronization Defects in Java Programs: Extended Static Analyses and Code Patterns. In: Proc. of 1st International Workshop on Multicore Software Engineering, pp 41–46
26. von Praun C (2004) Detecting Synchronization Defects in Multi-Threaded Object-Oriented Programs. PhD thesis, ETH Zurich
27. Ramalingam G (2000) Context-Sensitive Synchronization-Sensitive Analysis is Undecidable. ACM Trans Program Lang Syst 22(2):416–430
28. Shanbhag VK (2008) Deadlock-Detection in Java-Library Using Static-Analysis. In: Proc. of the 15th Asia-Pacific Software Engineering Conference, pp 361–368
29. Vallée-Rai R, Hendren L, Sundaresan V, Lam P, Gagnon E, Co P (1999) Soot - a Java Optimization Framework. In: Proc. of the 1999 conference of the Centre for Advanced Studies on Collaborative Research, pp 125–135
30. Williams A, Thies W, Ernst MD (2005) Static Deadlock Detection for Java Libraries. In: Proc. of the European Conference on Object-Oriented Programming, pp 602–629

## Appendix: Unexamined Alias Patterns

In this appendix, we formally justify the use of a Boolean encoding along with SAT/SMT solvers to perform the symbolic enumeration of unexamined alias patterns. Specifically, we justify the use of SAT to test for unexamined patterns in line 6 of Algorithm 4 by showing that the underlying problem of detecting unexamined alias patterns is NP-complete. Let $G_1 : (N_1, E_1)$ and $G_2 : (N_2, E_2)$ be two graphs. An aliasing pattern is a binary relation $\alpha \subseteq N_1 \times N_2$ between the nodes of $G_1$ and $G_2$. Recall that the execution of our algorithm for symbolic enumeration of "interesting" aliasing patterns yields the set $\mathscr{S}$ (set of aliasing patterns that are maximally safe) and the set $\mathscr{D}$ (set of aliasing patterns that are minimally unsafe). Also recall that in the set $\mathscr{S}$, maximally safe patterns are obtained by adding aliases to safe patterns as long as they do not cause deadlocks (cf. line 7 in Algo. 4). Similarly, minimally unsafe patterns are added to $\mathscr{D}$ by removing pairs of aliases from a deadlock causing pattern until no more can be removed (cf. line 14 in Algo. 4).

We say that a pattern $\alpha$ is *unexamined* w.r.t $\mathscr{S}, \mathscr{D}$ iff

$$(\forall S_i \in \mathscr{S}\ \alpha \not\subseteq S_i) \text{ AND } (\forall\, D_i \in \mathscr{D}\ D_i \not\subseteq \alpha)\ .$$

We now consider the problem AnyUnexaminedPatterns as below:

| | |
|---|---|
| **Inputs:** | $(G_1, G_2, \mathscr{S}, \mathscr{D})$ |
| **Output:** | YES, iff $\exists \alpha \subseteq N \times N$ *unexamined* w.r.t $\mathscr{S}, \mathscr{D}$. |
| | NO, otherwise. |

**Theorem A** *AnyUnexaminedPatterns is NP-complete.*

*Proof* Membership in NP is straightforward. An aliasing pattern $\alpha$ claimed to be unexamined can be checked by iterating over the aliasing patterns in $\mathscr{S}$ and $\mathscr{D}$, and checking (in polynomial time) for the subset relation.

We prove NP-hardness by reduction from the CNF satisfiability problem. Let $V = \{x_1, \ldots, x_n\}$ be a set of Boolean-valued variables and $\mathscr{C} = \{C_1, \ldots, C_m\}$ be a set of disjunctive clauses over literals of the form $x_i$ or $\neg x_i$. Corresponding to this instance of SAT, we create an instance $\langle G_1, G_2, \mathscr{S}, \mathscr{D} \rangle$ of the AnyUnexaminedPatterns problem.

Consider a graph $G_1$ consisting of $n$ vertices, each labeled with a variable in $V$. Consider a graph $G_2$ consisting of two vertices labelled $true$ and $false$, respectively. Informally, aliasing between the node labeled $x_i$ in $G_1$ and a node in $G_2$ can be interpreted as an assignment of $true$ or $false$ to $x_i$. We now design the sets $\mathscr{S}$ and $\mathscr{D}$ so that any unexamined aliasing pattern $\alpha$ has the following properties:

1. For each $x_i$, exactly one tuple in the set $\{(x_i, true), (x_i, false)\}$ belongs to $\alpha$. In other words, $\alpha$ represents an assignment of truth values to variables in $V$.
2. The assignment represented by $\alpha$ is a solution to the original SAT problem.

We define the set $\mathscr{S}$ as $\{A_1, \ldots, A_i, \ldots, A_n\}$, where

$$A_i :\ (V - \{x_i\}) \times \{true, false\}\,.$$

Intuitively, each $A_i$ represents an aliasing pattern in which the $x_i$ variable is missing, and all other variables have both the $true$ and $false$ value assigned. Clearly, any unexamined pattern that is a subset of any $A_i$ does not have a truth-value assigned to the variable $x_i$, and hence cannot represent a valid assignment of truth values to the original SAT problem.

We define the set $\mathscr{D}$ as a union of two sets $B$ and $T$. The set $B$ is defined as $\{B_1,\ldots,B_n\}$, where:

$$B_i:\ \{(x_i,true),(x_i,false)\}\,.$$

Intuitively, any aliasing pattern $\alpha$ that is a superset of some $B_i$ cannot represent a valid truth value assignment to the original SAT instance, as it would contain conflicting assignments of truth values to the variable $x_i$.

The set $T$ is defined in terms of the clauses $C_i \in \mathscr{C}$. $T = \{T_1,\ldots,T_m\}$, wherein $T_i$ corresponds to the $i^{th}$ clause $C_i$ as follows:

$$T_i = \bigcup_j \begin{cases} \{(x_j,false)\} & x_j \in C_i \\ \{(x_j,true)\} & \neg x_j \in C_i \end{cases}$$

Intuitively, any aliasing pattern $\alpha$ that is a superset of $T_i$ cannot satisfy the clause $C_i$ (*i.e.*, $C_i = false$). Hence, such an $\alpha$ cannot represent a solution to the SAT problem. Combining $B$ and $T$, any $\alpha$ that is the superset of any aliasing pattern $D_i \in \mathscr{D}$, thus, cannot represent a solution to the original SAT problem.

To summarize, corresponding to each SAT instance $(V,C)$, we construct an instance of the AnyUnexaminedPatterns problem with

$$\mathscr{S}:\ \{A_1,\ldots,A_n\}\text{, and, }\ \mathscr{D}:\ \{B_1,\ldots,B_n\}\ \cup\ \{T_1,\ldots,T_m\}$$

In order to complete the proof, we show that there is a satisfying assignment to the original problem if and only if there is an unexamined aliasing pattern.

Let $\mu : \{x_1,\ldots,x_n\} \mapsto \{true,false\}$ be any satisfying solution to the original problem. We construct a pattern $\alpha$ that maps $x_i$ to $true$ if $\mu(x_i) = true$ and to $false$ otherwise. We now show that $\alpha$ is an unexamined aliasing pattern. It is easy to see that $\alpha \not\subseteq A_i$, since $\alpha$ contains at least one of $(x_i,true)$ or $(x_i,false)$. We can also show that $B_i \not\subseteq \alpha$ since $\alpha$ contains only consistent assignments for each variable $x_i$. Similarly, $T_i \not\subseteq \alpha$, or else the corresponding clause $C_i$ is not satisfied by $\alpha$. Therefore $\alpha$ is an unexamined aliasing pattern. Conversely, we can demonstrate that any unexamined aliasing pattern $\alpha$ that can be discovered corresponds to a satisfying truth assignment. This shows that the problem AnyUnexaminedPattern can be obtained as a reduction from CNF-SAT, and is thus NP-complete.