Symbolic Deadlock Analysis in Concurrent Libraries and their Clients

Jyotirmoy Deshmukh
Univ. of Texas at Austin.
jyotirmoy@cerc.utexas.edu

E. Allen Emerson
Univ. of Texas at Austin.
emerson@cs.utexas.edu

Sriram Sankaranarayanan
Univ. of Colorado at Boulder.
srirams@colorado.edu

Abstract

Methods in object-oriented concurrent libraries hide internal synchronization details. However, information hiding may result in clients causing thread safety violations by invoking methods in an unsafe manner. Given such a library, we present a technique for inferring interface contracts that specify permissible concurrent method calls and patterns of aliasing among method arguments, such that the derived contracts 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 for tracking lock dependencies, allowing us to synthesize contracts using a SMT solver. 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 the client code.*

1. Introduction

Concurrent programs are prone to a variety of thread-safety violations arising from the presence of *data race conditions* and *deadlocks*. In practice, data races are abundant and difficult to debug. Thus, 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 lock acquisition. 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 shared resources such as locks. Deadlock detection is a well-studied problem, and both static and dynamic approaches have been proposed [1–7]. 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 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. [7]. 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. [7], 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:

(a) We reason about possible aliasing patterns between nodes in a lock graph explicitly rather than use types as 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 be-

```
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: Methods in java.awt.EventQueue

tween 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.

1.1. Approach at a Glance

To illustrate the problem with the standard deadlock analysis techniques (*i.e.* treating libraries as closed systems), we use the Java code snippet (shown in Fig. 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 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 [8] (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 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: ob_1, ob_2:

$$T_1 : ob_1.\texttt{wakeup(true)} \;||$$
$$T_2 : ob_2.\texttt{postEventPrivate}()$$

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. 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 $ob_{1,2}$ are not aliased/do not share fields. Consider, on the other hand, the scenario wherein the object ob_1.nextQueue aliases ob_2 and ob_2.nextQueue aliases ob_1. Under such a scenario, the lock-order graph of Fig. 2 is modified by fusing the aliased nodes into a single node to obtain the graph depicted in Fig. 3.

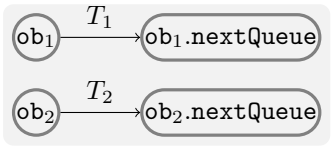


Fig. 2: Joint lock-order graph from the postEventPrivate and wakeup methods.

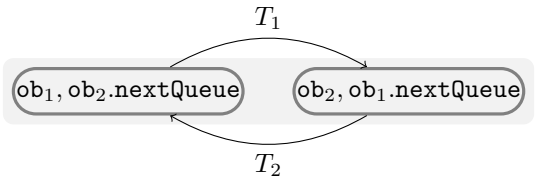


Fig. 3: Lock-order graph for $T_1||T_2$ under aliasing of nodes.

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 [8] (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. 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/sharing scenarios between objects at the concurrent call-sites of library methods,

(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: ¬isAliased(`a,b.nextQueue`) ∨ ¬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. Experimental results obtained by analyzing well-known `Java` libraries are discussed in Sec. 5. In Sec. 6, we show how interface contracts derived for a given library can be used compositionally while analyzing the client of the library for deadlocks. Finally, we discuss related work, and conclude in Sec. 7.

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 $\mathcal{L}$ as a collection of *class* definitions $\langle \mathcal{C}_1, \ldots \mathcal{C}_k \rangle$. Each class $\mathcal{C}_i$ 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 $\mathcal{C}_2$ is a subclass of $\mathcal{C}_1$.

Data members may have primitive types (`int`, `double`, *etc.*), or reference types $C_1, \ldots, C_k$[1]. An object is an instance of a class C_i (of a non-primitive type). 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 $\mathcal{L}$.

Def. 2.1 (Access Expressions)**.** Given an 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) If e_j is an access expression of non-primitive type C_j, and f_k a field of the class C_j of type C_k. Then $\mathbf{e}: e_j.f_k$ is also an access expression of type C_k.

1. Apart from class types, array types are also classified as reference types, and while our technique does handle array variables conservatively, we omit a detailed discussion on array types for brevity.

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.

Def. 2.2 (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_i of class $\mathcal{C}$ is associated with a signature $sig(m_i)$ that defines the types for the formal parameters of m_i, and a return type. Every method m_i is always executed on an object of some type C_i. A method body consists of a sequence of statements, including calls to other member methods of classes within $\mathcal{L}$. The operational semantics of m are defined using a control-flow graph (denoted $cfg(m)$). We define $cfg(m)$ as a tuple (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 $s \in S$.

Lock-based synchronization: We seek to analyze object libraries that support concurrent accesses to their fields and methods. Therefore synchronization statements such as *lock-acquisition* and *lock-release* of the form `lock`(`ob`) and `unlock`(`ob`) are used to serialize accesses to critical regions. Following the `Java` convention, we assume an associated monitor for every object `ob`. A thread executing `lock`(`ob`) is blocked unless it can successfully acquire the monitor associated with `ob`. The statement `unlock`(`ob`) releases the monitor, returning it to the unlocked state. Note that in `Java`, the syntax of a `synchronized`$(ob)\{\cdots\}$ statement ensures that lock acquisition and release are matched. Other high-level programming languages also permit synchronization constructs based on *semaphores* and *rendezvous*. However, in this paper we focus on deadlocks arising from purely lock-based synchronization. In practice, lock-based synchronization accounts for a vast majority of synchronization operations used in `Java` programs, as well as for most of the bugs caused due to their misuse. We use a *lock-order graph* to represent patterns of nested lock accesses for a given method.

Def. 2.3 (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.

Static Computation of $lg(m)$: The standard interprocedural approach to compute the lock-order graph for the methods of a given library involves the summarization of each method within the library. A summary $sum(s, m_i)$ is the symbolic state of m_i after executing a program statement s, described as a data structure: (lg, ls, rs, env), where lg is the lock-order graph, ls is the set of locks acquired (but not released) by m_i, rs is the set of locks that do not have any predecessors (root nodes), and env is a mapping that tracks any local variables that may be aliased to global variables on the heap, and thus escape the scope of m_i.

We closely follow the technique described in [7] for fixpoint-based summary computation at each statement. Briefly, for a statement s that acquires a new lock l, we add edges from every lock in ls to l, and then add l to ls. Releasing a lock l corresponds to removing l from ls. A branch-merge corresponds to computing $lg_1 \cup lg_2$, where each lg_i is the lg computed along the i^{th} branch. Upon encountering a call to a method m_j, we concatenate $lg(m_j)$ to lg at the invocation point, while replacing the formal parameters in $lg(m_j)$ with the actual parameters at the call-site. 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. We ensure termination by artificially bounding the size of access expressions considered by our approach.

3. Approach

Let $m_1, \ldots, m_k$ be a set of methods in library $\mathcal{L}$ 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 $\texttt{ob}_1, \ldots, \texttt{ob}_k$ denote a set of objects on which the methods $m_1, \ldots, m_k$ are invoked. Let $\texttt{ob}_{k+1}, \ldots, \texttt{ob}_n$ 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 variables $\texttt{ob}_1, \ldots, \texttt{ob}_n$ in lieu of the "`this`" object and the parameters of m_1 and m_2. We assume that $lg(m_1)$ and $lg(m_2)$ are themselves cycle free. Let $V_{1,2} = \{\texttt{e}_1, \ldots, \texttt{e}_m\}$ denote the set of access expressions occurring in $lg(m_1) \cup lg(m_2)$. We first characterize the patterns of aliasing/sharing between the access expressions corresponding to $\texttt{ob}_1, \ldots, \texttt{ob}_n$ under some fixed runtime environment R.

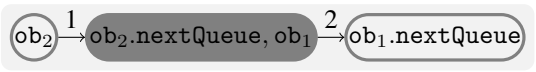
Def. 3.1 (Aliasing Pattern)**.** An aliasing pattern α over a set of access expressions V is a symmetric, reflexive and transitive relation over V such that if $(e_1, e_2) \in \alpha$, $(e_1.f_i, e_2.f_i) \in \alpha$ for all shared fields f_i between $\mathsf{Type}(e_1)$ and $\mathsf{Type}(e_2)$.

Given an aliasing pattern α over the nodes of a lock-order graph G, 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 $\alpha \triangleright G$ denote the resulting graph after merging all aliased nodes.

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

Example 3.1. Consider methods m_1 (`wakeup`) and m_2 (`postEventPrivate`) from the `java.awt.EventQueue` class, as shown in Fig. 1. Sec. 1 illustrates the individual lock-order graphs $lg(m_1)$ and $lg(m_2)$. Following the notation established, let $\texttt{ob}_1, \texttt{ob}_2$ denote the objects on which methods m_1, m_2 are invoked, respectively. The access expressions involved in the lock graph $G : lg(m_1) \cup lg(m_2)$ are $V_{1,2} = \{\texttt{ob}_1, \texttt{ob}_2, \texttt{ob}_1\texttt{.nextQueue}, \texttt{ob}_2\texttt{.nextQueue}\}$. Let α_1 be the aliasing pattern $\{(\texttt{ob}_1, \texttt{ob}_2\texttt{.nextQueue})\}$. The merged lock graph $\alpha_1 \triangleright G$ is shown below:

$\alpha_1 \triangleright G$:



The pattern α_1 does not cause a deadlock. However, the pattern $\alpha_2 : \{(\texttt{ob}_2, \texttt{ob}_1\texttt{.nextQueue}), (\texttt{ob}_1, \texttt{ob}_2\texttt{.nextQueue})\}$ considered in Sec. 1 is deadlock-causing.

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

A simplistic approach consists of (a) enumerating all possible aliasing patterns α, and (b) checking every graph $\alpha \triangleright G$ for a cycle. As pointed out in [7], 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 using SAT-Modulo Theory (SMT) solvers to perform the enumeration efficiently.

3.1. Symbolic Encoding

We first discuss how we encode the cycle detection problem, given a graph $G = lg_1 \cup lg_2$, and a fixed aliasing pattern α, into an efficient theory amenable to a SMT solver. 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 α 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 $v_i \rightarrow v_j$ 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:

$$\Psi(G) : \bigwedge_{(v_i,v_j)\in E} (\mathbf{x}(\mathbf{v_i}) < \mathbf{x}(\mathbf{v_j})) . \quad (1)$$

Example 3.2. Consider once again the running example from Fig. 1, continuing with the notation established in Ex. 3.1. The merged lock graph $G : \ lg(m_1) \cup lg(m_2)$ is recalled in Fig. 2. The constraint $\Psi(G)$ for this graph is as follows:

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

Aliasing Pattern Encoding : Given an aliasing pattern α, 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 α by means of a set of equalities as follows:

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

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 α, and a graph G, the formulae $\Psi(G)$ and $\Psi(\alpha)$ 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 in accordance with the aliasing pattern α.

Example 3.4. Continuing with Ex. 3.3, we recall $\Psi(G)$ from Ex. 3.1, below:

$$\Psi(G) : x(\mathtt{ob}_1) < x(\mathtt{ob}_1.\mathtt{nextQueue}) \wedge$$
$$x(\mathtt{ob}_2) < x(\mathtt{ob}_2.\mathtt{nextQueue}).$$

and $\Psi(\alpha_1) : \ x(\mathtt{ob}_2) = x(\mathtt{ob}_1.\mathtt{nextQueue})$. The combined 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 $\alpha_2 : \ \{(\mathtt{ob}_2, \mathtt{ob}_1.\mathtt{nextQueue}), (\mathtt{ob}_1, \mathtt{ob}_2.\mathtt{nextQueue})\}$:

$$\Psi(\alpha_2) : x(\mathtt{ob}_2) = x(\mathtt{ob}_1.\mathtt{nextQueue}) \wedge$$
$$x(\mathtt{ob}_1) = x(\mathtt{ob}_2.\mathtt{nextQueue}).$$

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 α_2 may cause a deadlock.

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

Constraint Solving: Given an aliasing pattern α, 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 [9]. In practice, solving Boolean combinations of UTVPI and equality constraints can be solved quite efficiently using modern SMT solvers such as Yices and Z3 [10, 11].

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 $\mathtt{e}_i \in V$ *cannot* be removed without missing any potential deadlocks.

Example 4.1. Returning to the lock graph in Fig. 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 $\mathsf{mayAlias}(v) = \{u \in V |\ u \text{ may-alias } v\}$.

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.

Def. 4.1 (Admissible)**.** An aliasing pattern α 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 α adding more aliases to α does not remove the deadlock. Similarly, for a safe pattern β, removing aliases from β does not cause a deadlock.

Def. 4.2 (Subsumption)**.** A pattern α_2 *subsumes* α_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, α_1 is a sub-relation of α_2.

Lemma 4.1. *If α_1, α_2 are aliasing patterns, and $\alpha_1 \subseteq \alpha_2$, then the following are $true$:*

(A) α_1 is deadlock-causing $\Rightarrow$ α_2 is deadlock-causing,
(B) α_2 is safe $\Rightarrow$ α_1 is safe.

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

Def. 4.3 (Maximally Safe/Minimally Unsafe)**.** A pattern α 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 α 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 α that is *safe*, we can add previously unaliased pairs of aliases to α 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 β, we remove "unnecessary" alias pairs from β 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. 2 shows the resulting graph. We can add the pair $(\texttt{ob}_1, \texttt{ob}_2\texttt{.nextQueue})$ to α_0 without creating any cycles. The resulting pattern α_1 is shown in Ex. 3.1. However, if the pair $(\texttt{ob}_2, \texttt{ob}_1\texttt{.nextQueue})$ were added to α_1 then we obtain a cycle in the graph. As a result, the pattern α_1 is maximally safe.

The explicit enumeration algorithm (Algorithm 1) 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 α 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 α as long as the addition does not create a cycle in $\alpha' \triangleright G$, where α' is the symmetric and transitive closure of $\alpha \cup \{(u, v)\}$. The result is a pattern α that is maximally safe, which is then added to the set $\mathcal{S}$ (Line 9). We then remove all patterns β that are subsumed by α 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 α 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 [2] (Line 13). The set U of unexamined patterns is pruned by removing all patterns that subsume α' (such patterns also cause a deadlock) (Line 14).

The application of Algo. 1 on the graph from Fig. 2 enumerates the max. safe/ min. unsafe patterns in Table 1.

2. Note that α' may not be a minimally unsafe relation.

Algorithm 1: EnumerateAllAliasingPatterns

Input: G : Graph
Result: $\mathcal{D}$: 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 $\alpha' :=$ Closure$(\{(u, v)\} \cup \alpha)$
8 **if** $\alpha' \triangleright G$ *is acyclic* **then** $\alpha := \alpha'$
/* α *maximally safe* */
9 $\mathcal{S} := \mathcal{S} \cup \{\alpha\}$
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 $\alpha' := \alpha \cap \{(u, v) \mid u, v \in C\}$
/* α' is unsafe */
14 $U := U - \{\beta \mid \alpha' \subseteq \beta\}$
15 $\mathcal{D} := \mathcal{D} \cup \{\alpha'\}$
16
17 **end**

Algorithm 2: SymbolicEnumerateAllAliasingPatterns

Input: G : Graph
Result: $\mathcal{D}$: Deadlock Scenarios

1 **begin**
2 $\Psi_U :=$ $\Psi_0(V)$ (encoding all alias patterns)
3 **while** Ψ_U *SAT* **do**
4 $(y(\mathsf{e}_1), \ldots, y(\mathsf{e}_k)) :=$ Solution of Ψ_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 $\alpha' :=$ 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** $\alpha := \alpha'$
/* α 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 $\mathcal{S} := \mathcal{S} \cup \{\alpha\}$
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 $C\}$
/* α' 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 $\mathcal{D} := \mathcal{D} \cup \{\alpha'\}$
18
19 **end**

TABLE 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

Symbolic Enumeration Algorithm.

Algorithm 1 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 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 e_i. We then encode all aliasing patterns with the help of a logical formula Ψ_0 involving the $y(\mathsf{e}_i)$ variables, as follows: $\Psi_0(V) =$

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

The formula Ψ_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 2 shows the symbolic version of Algorithm 1. The correspondence between the two algorithms is immediately observable upon 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 α (Line 5). Recall from Sec. 3.1 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 α (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 α (Line 7). Once a maximally safe α is obtained, we remove all aliasing patterns that are subsumed by α from the set of all aliasing patterns (represented by Ψ_U), and add α to $\mathcal{S}$ (Line 12). If the formula is unsatisfiable, then we obtain the minimal unsatisfiable core (Line 14) and extract the minimally unsafe aliasing pattern α' from the constraints represented in this core. We then remove all aliasing patterns that subsume α' from Ψ_U (Line 16), and add the minimally unsafe α' obtained (if any) to the set $\mathcal{D}$ (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 [10, 11]. This is very useful in the context of Algorithm 2. 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. 1 and Algo. 2 can generate a contract that *succinctly* represents the set of all safe aliasing patterns. The result of the enumeration is a set of patterns $\mathcal{D}$ such that any aliasing pattern β is deadlock-causing iff it subsumes a pattern $\alpha \in \mathcal{D}$.

Lemma 4.2. *An alias pattern α is safe iff for all $\beta \in \mathcal{D}$, $\beta \not\subseteq \alpha$.*

In practice, contract derivation consists of first *compacting* the set $\mathcal{D}$ 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 $\mathcal{D}$.

Example 4.3. From Table 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:

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

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 $\mathcal{D}$ of deadlock-causing alias patterns for each pair of library methods obtained by the enumeration technique in Algo. 2 yields a* contract *of the form: $\bigwedge_{\alpha \in \mathcal{D}} \bigvee_{(e_i, e_j) \in \alpha} \neg \textit{isAliased}(e_i, e_j)$.*

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. 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 [12]. Must-aliases for lock objects are tracked across methods using our own analyzer built using the `soot`'s intraprocedural alias analysis (`spark`). We also augment `spark` in order to track aliasing between fields, yielding a field-sensitive 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).
(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 an *escape analysis*.

These filtering strategies are sound: our tool does not miss any potential deadlock due to these strategies. The generated constraints are solved using the SMT solver Yices [10]. Table 2 summarizes the potential deadlocks thus obtained. Table 2 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). A detailed presentation of our benchmarks and results is available online [3].

Example 5.1. From the lock-order graph of the `postEventPrivate` method presented in Sec. 1.1, we can see that the concurrent invocation of `a.postEventPrivate(...)` and `b.postEventPrivate(...)` leads to a deadlock under a specific aliasing pattern. However, the methods `postEvent`, `push` and `pop` in the same class also invoke the

3. cf. http://cerc.utexas.edu/~jyotirmoy/deadlock/ for more details.

TABLE 2: Experimental Results

Library	KLOC	Num. of Aliasing Patterns Examined	Num. of DL-causing Patterns	Time taken (secs)[a]		Num. of Unique Scenarios	
				Summary Computation	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.

`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`.

Library Name	Method names	Bug Report
java.awt (EventQueue)	postEventPrivate, wakeup	[8]:4913324 [8]:6424157, [8]:6542185
java.awt (Container)	removeAll, addPropertyChangeListener	[13]
java.util (LogManager) (Logger)	addLogger getLogger	[8]:6487638
javax.swing (JComponent)	setFont paintChildren	Jajuk [14]
hsqldb (Session)	isAutoCommit close	[15]

TABLE 3: Real Client Deadlocks

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 3 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.

6. 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 [16] 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* [6, 17]. 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.

7. Related Work and Conclusions

Runtime Techniques : Runtime techniques for deadlock detection track nested lock-acquisition patterns. The *GoodLock* algorithm [1] is capable of detecting deadlocks arising from two concurrent threads; [18] 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. [2] further extend this approach to programs with semaphores and condition variables.

Model Checking : Model checking techniques [19] have been successfully used to detect deadlocks in programs [20]. For instance, Corbett et al. employ model checking to analyze protocols written in Ada for deadlocks [3]. Model checkers such as *SPIN* [21], *Java Path Finder* [1, 22] have been used extensively to check concurrent `Java` programs for deadlocks. However, program size and complexity limit these approaches. A compositional technique based on summarizing large libraries can help these approaches immensely. Bensalem et al. [23] 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 [4, 24] 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 [6]. Most static techniques focus on identifying deadlocks within a given closed program, while in [6], the authors close a given open program (the library) by manually constructing a harness for that program. In [26] 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. [7] 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.

Conclusions. 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.

Our approach is currently limited to lock-based synchronization for re-entrant locks, but is also applicable to libraries with conditional synchronization with monitors (`wait`/`notify` constructs in `Java`), semaphores, and locks with arbitrary re-entrancy models, and remains an important part of the future work. 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 [23], 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 `pthreads` 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. 6), 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] K. Havelund, "Using runtime analysis to guide model checking of java programs," in *Proc. of SPIN Workshop on Model Checking of Software*, 2000, pp. 245–264.

[2] R. Agarwal and S. D. Stoller, "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*, 2006, pp. 51–60.

[3] J. C. Corbett, "Evaluating deadlock detection methods for concurrent software," *IEEE Transactions on Software Engineering*, vol. 22, no. 3, pp. 161–180, 1996.

[4] C. von Praun, "Detecting synchronization defects in multi-threaded object-oriented programs," Ph.D. dissertation, ETH Zurich, 2004.

[5] C. Artho and A. Biere, "Applying static analysis to large-scale, multi-threaded java programs," in *Proc. of the 13th Australian Conference on Software Engineering*, 2001, p. 68.

[6] M. Naik, C.-S. Park, K. Sen, and D. Gay, "Effective static deadlock detection," in *Proc. of the 31st International Conference on Software Engineering*, 2009, pp. 386–396.

[7] A. Williams, W. Thies, and M. D. Ernst, "Static deadlock detection for java libraries," in *Proc. of the European Conference on Object-Oriented Programming*, 2005, pp. 602–629.

[8] "Sun developer network bug database," 2007, bug-id provided at citation. [Online]. Available: http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=xxxx

[9] S. K. Lahiri and M. Musuvathi, "An efficient decision procedure for UTVPI constraints," in *Proc. of Frontiers of Combining Systems, 5th International Workshop*, 2005, pp. 168–183.

[10] B. Dutertre and L. de Moura, "A fast Linear-Arithmetic solver for DPLL(T)," in *Proc. of Computer Aided Verification*, 2006, pp. 81–94.

[11] L. de Moura and N. Bjϕrner, "Z3: An efficient SMT solver," in *Proc. of Tools and Algorithms for the Construction and Analysis of Systems*, 2008, pp. 337–340.

[12] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co, "Soot - a java optimization framework," in *Proc. of the 1999 conference of the Centre for Advanced Studies on Collaborative Research*, 1999, pp. 125–135.

[13] *Open Source Mail Archive, Bug 159*, 2008. [Online]. Available: http://osdir.com/ml/java.openjdk.distro-packaging.devel/2008-06/msg00061.html

[14] *Jajuk Bug Ticket*. [Online]. Available: http://trac.jajuk.info/ticket/850

[15] *Open Source Mail Archive*, 2004. [Online]. Available: http://osdir.com/ml/java.hsqldb.user/2004-03/msg00150.html

[16] L. Li and C. Verbrugge, "A practical MHP information analysis for concurrent Java programs," in *Proc. of the 17th International Workshop on Languages and Compilers for Parallel Computing*, 2004, pp. 194–208.

[17] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for java," in *Proc. of the 2006 ACM SIGPLAN conf. on Programming Language Design and Implementation*. ACM, 2006, pp. 308–319.

[18] R. Agarwal, L. Wang, and S. Stoller, "Detecting potential deadlocks with static analysis and run-time monitoring," *Hardware and Software, Verification and Testing*, pp. 191–207, 2006.

[19] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching-time temporal logic," in *Logics of Programs*, 1981, pp. 52–71.

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

[21] G. J. Holzmann, *The SPIN model checker*. Addison-Wesley, 2003.

[22] K. Havelund and T. Pressburger, "Model checking JAVA programs using JAVA PathFinder," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 366–381, Mar. 2000.

[23] S. Bensalem and K. Havelund, "Dynamic deadlock analysis of multi-threaded programs," in *Proc. of the Haifa Verification Conference*, 2005, pp. 208–223.

[24] D. Engler and K. Ashcraft, "Racerx: Effective, static detection of race conditions and deadlocks," *ACM SIGOPS Operating System Review*, vol. 37, no. 5, pp. 237–252, Dec. 2003.

[25] F. Otto and T. Moschny, "Finding synchronization defects in java programs: Extended static analyses and code patterns," in *Proc. of 1st International Workshop on Multicore Software Engineering*, 2008, pp. 41–46.

[26] V. K. Shanbhag, "Deadlock-detection in java-library using static-analysis," in *Proc. of the 15th Asia-Pacific Software Engineering Conference*, 2008, pp. 361–368.

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 2 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 $\mathcal{S}$ (set of aliasing patterns that are maximally safe) and the set $\mathcal{D}$ (set of aliasing patterns that are minimally unsafe). Also recall that in the set $\mathcal{S}$, maximally safe patterns are obtained by adding aliases to safe patterns as long as they do not cause deadlocks (cf. line 7 in Algo. 2). Similarly, minimally unsafe patterns are added to $\mathcal{D}$ by removing pairs of aliases from a deadlock causing pattern until no more can be removed (cf. line 14 in Algo. 2).

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

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

We now consider the problem AnyUnexaminedPatterns as below:

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

Theorem A. *AnyUnexaminedPatterns is NP-complete.*

Proof: Membership in NP is straightforward. An aliasing pattern α claimed to be unexamined can be checked by iterating over the aliasing patterns in $\mathcal{S}$ and $\mathcal{D}$, 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 $\mathcal{C} = \{C_1, \ldots, C_m\}$ 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 $\langle G_1, G_2, \mathcal{S}, \mathcal{D} \rangle$ 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 $\mathcal{S}$ and $\mathcal{D}$ so that any unexamined aliasing pattern α has the following properties:

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

We define the set $\mathcal{S}$ 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 $\mathcal{D}$ 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 α 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 $C_i \in \mathcal{C}$. $T = \{T_1, \ldots, T_m\}$, 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 α that is a superset of T_i cannot satisfy the clause C_i (*i.e.*, $C_i = false$). Hence, such an α cannot represent a solution to the SAT problem. Combining B and T, any α that is the superset of any aliasing pattern $D_i \in \mathcal{D}$, 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

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

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 α that maps x_i to $true$ if $\mu(x_i) = true$ and to $false$ otherwise. We now show that α is an unexamined aliasing pattern. It is easy to see that $\alpha \not\subseteq A_i$, since α contains at least one of $(x_i, true)$ or $(x_i, false)$. We can also show that $B_i \not\subseteq \alpha$ since α 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 α. Therefore α is an unexamined aliasing pattern. Conversely, we can demonstrate that any unexamined aliasing pattern α 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.

□