Modular Verification of Synchronization with Reentrant Locks

Tevfik Bultan
Department of Computer Science
University of California
Santa Barbara, CA, USA
bultan@cs.ucsb.edu

Fang Yu
Department of Computer Science
University of California
Santa Barbara, CA, USA
yuf@cs.ucsb.edu

Aysu Betin Can
Informatics Institute
Middle East Technical University
Ankara, Turkey
aysu@ii.metu.edu.tr

***Abstract*—We present a modular approach for verification of synchronization behavior in concurrent programs that use reentrant locks. Our approach decouples the verification of the lock implementation from the verification of the threads that use the lock. This decoupling is achieved using lock interfaces that characterize the allowable execution order for the lock operations. We use a thread modular verification approach to check that each thread obeys the lock interface. We verify the lock implementation assuming that the threads behave according to the lock interface. We demonstrate that this approach can be used to verify synchronization behavior in Java programs that use reentrant lock implementations for synchronization.**

I. Introduction

Concurrent programming is a difficult task since it requires coordinating execution of multiple threads. It is especially difficult to control the behavior of shared variables that can be accessed and modified by multiple threads. A common concurrent programming paradigm is to protect access to shared variables using *locks*. Locks restrict access to shared variables and enable the threads that hold the lock to read and modify shared variables without interference from other threads. However, for this paradigm to work, there are two correctness criteria that have to be satisfied: 1) The locks have to be used correctly by the threads, and 2) The locks have to be implemented correctly. In this paper, we present a modular verification approach that can be used to check both of these conditions.

Reliable concurrent programming is a challenge faced by all Java programmers since threads are an integral part of Java. Thread programming in Java requires conditional waits and notifications implemented with multiple locks and multiple condition variables with associated `synchronized`, `wait`, `notify` and `notifyAll` statements. Concurrent programming using these synchronization primitives is error-prone, with common programming errors such as nested monitor lockouts, missed or forgotten notifications, slipped conditions, etc. [14].

Reentrant locking is an important concurrent programming concept for fine grain synchronization. As part of J2SE 5.0, `java.util.concurrent` package [20] contains various lock implementations to help Java programmers with synchronization. For example, the `ReentrantLock` and `ReentrantReadWriteLock` are lock implementations provided by the `java.util.concurrent` package, and these locks are reentrant, i.e., if a thread acquires a reentrant lock and tries to acquire the same lock again, then the thread would not block. The default synchronization mechanism in Java based on the `synchronized` keyword is also reentrant. A reentrant lock is released by a thread when the number of acquire and release operations executed by that thread become equal.

In this paper, we present a modular verification approach for verification of reentrant lock implementations and their usage. Our modular verification approach is based on the earlier work on verification of synchronization policies in Java programs presented in [2]–[5]. However, these earlier results are not capable of handling reentrant locks since they restrict the interfaces of the synchronization policies to finite state machines. A finite state machine is not expressive enough for specifying the interface of a reentrant lock. Consider a reentrant lock with a single acquire (a) and a single release (r) operation. The lock is released by a thread when the sequence of calls made by that thread is in $\{a^{i_1}r^{i_1}a^{i_2}r^{i_2}\ldots a^{i_n}r^{i_n} \mid i_1, i_2, \ldots, i_n, n \geq 0\}$. This constraint cannot be modeled using finite state machines. To express this type of interface constraints we use extended finite state machines and keep track of the difference between acquire and release operations for each lock. Based on these lock interfaces, we develop a modular verification technique that enables verification of both the implementation and the usage of reentrant locks such as `ReentrantLock` and `ReentrantReadWriteLock`. Although we focus on verification of concurrent Java programs in this paper, our approach is also applicable to other concurrent programming frameworks that support reentrant locks.

Figure 1 gives an overview of our verification framework. First we separate the verification task to two main components: 1) Checking the lock behaviors, i.e., making sure that the lock implementations are correct. 2) Checking that the threads obey the lock interfaces. A lock interface specifies the allowable sequences of lock and shared operations. As mentioned above, in our framework the lock interfaces are

This work is supported by NSF grants CCF-0916112 and CCF-0716095.

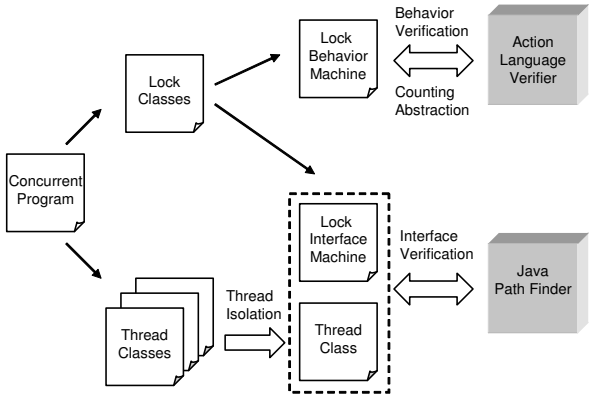


Figure 1. Overview of the framework

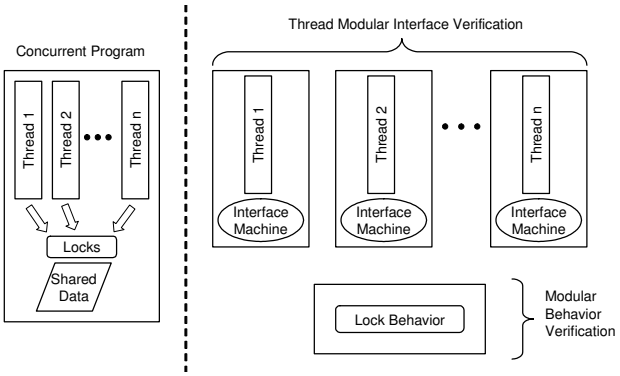


Figure 2. Modular lock verification

specified as extended state machines.

In order to check a lock's behavior we assume that the threads that use the lock obey the lock's interface. This allows us to check the lock's behavior by replacing the threads with instances of the state machine that characterizes the lock's interface. In our framework the lock behavior is specified by a set of guarded commands that update the lock variables. We construct a lock behavior machine based on these guarded commands and the lock interface state machine.

One of our contributions in this paper is reducing the verification of a reentrant-lock behavior machine to verification of a non-reentrant-lock behavior machine. I.e., we show that during the lock behavior verification, it is not necessary to keep track of how many times a thread acquired or released a lock as long as it is holding the lock.

As shown in Figure 2, our approach is modular in two different dimensions. First, using the lock interfaces we are able to separate the lock behavior from the rest of the program. This is crucial for achieving scalable verification of lock properties. This separation and the reduction from reentrant-locks to non-reentrant-locks enable us to use the counting abstraction technique and check the lock implementations with respect to arbitrary number of threads. We use the Action Language Verifier (ALV) [22] for lock behavior verification.

Second dimension of modularity is the thread modular interface verification. We check interface correctness of each thread separately. We isolate the threads by replacing shared data variables with nondeterministic stubs that return arbitrary values. We replace the lock implementations with stubs that check that the thread that is being verified calls the lock operations in the order specified by the lock interface. We use the Java Path Finder (JPF) [21] for interface verification.

Related Work: This paper builds on the framework developed in [2]–[5]. The main contribution of the current paper is extending the modular verification framework introduced in this earlier work to reentrant locks. In order to achieve this, we make the following contributions: 1) We present a new formal model that captures the semantics of reentrant locks. This formal model partitions the lock operations to acquire and release operations and augments the concurrent program states and the interface machine states with reentry-counts that keep track of the difference between the acquire and release calls by each thread. 2) During interface verification we use these extended state machines augmented with reentry-counts to check the ordering of the lock and shared operations. 3) We show that, during behavior verification, it is not necessary to keep track of reentry-counts, i.e., we show that the behavior verification for reentrant-locks can be reduced to behavior verification for non-reentrant-locks. This enables us to use counting abstraction and check the lock implementations with respect to arbitrary number of threads. 4) We conduct experiments on Java programs with reentrant locks demonstrating the feasibility of the proposed modular verification approach.

The thread-modular reasoning discussed in [11] verifies each thread separately with respect to safety properties. The effects of other threads are modeled as environment assumptions whereas we use shared data stubs to reflect these effects. Besides, we check the thread behavior against the lock interfaces and leave the assurance of the safety properties to behavior verification. I.e., our approach uses a second type of modularity (which we call behavior-modularity) in addition to thread-modularity.

Context-bounding [15]–[17], i.e., limiting the number of context-switches among concurrent threads, is another way of overcoming intractability of concurrent program verification. Our approach relies on both thread and behavior modular verification rather than bounding the context switches.

Using state machines to characterize interfaces have been proposed in the past. In [6] interfaces of software modules are specified as a set of constraints, and algorithms for interface compatibility checking are presented. In [7] type systems are extended with stateful interfaces and interface checking is made part of type checking. We use interfaces to characterize the allowed call sequences for the lock and

shared operations and our interface machine characterization is tailored towards reentrant locks.

In [13] a methodology is proposed for preserving object invariants in a concurrent program with fine grained locking mechanism through an ownership system supported by a modular verification methodology. However, the approach proposed in [13] cannot handle synchronization policies such as the read-write lock. In our approach the programmers are able to implement customized locks and check their correctness.

In [1], an assertional proof method for proving properties of a multithreaded sublanguage of Java is presented by incorporating Java's reentrant synchronization mechanism to a proof system. Unlike our approach, this proof system requires annotation of programs with Hoare-style assertions. Furthermore, automation of the verification task is not discussed.

In [12] a verification technique for a concurrent Java-like language with reentrant locks is proposed. The proposed approach is based on separation logic. The authors introduce a notion of a lockset to handle reentrancy (in contrast to the reentry-counts used in our approach). Automation of the verification task is not discussed in this work either.

Finally, in [18], [19] a design-for-verification approach is presented that decouples the synchronization concerns from the functional logic of the program. This is achieved by using a language-independent compositional model for synchronization contracts. Although, the approach presented in [18], [19] uses the design-for-verification concept to achieve effective verification as we do in our work, it does not use the thread and behavior-modularity that we use in our approach. Moreover, the approach presented in [18], [19] requires the use of a middleware framework for contract negotiation at run-time, whereas our approach does not require any runtime intervention.

II. Modeling Concurrent Programs

In this section, we describe a simplified formal model for concurrent programs. We assume that a concurrent program consists of an arbitrary but fixed number of concurrent threads. Each thread has its local variables that can only be accessed and modified by that thread. The shared variables and locks are accessible to all threads. We keep locks separate from the shared variables since they are specifically used for synchronization purposes. We assume that the shared variables and locks can be modified only by executing their methods. (This can be achieved by making fields of shared and lock objects private.) We do not consider thread creation or termination in our model. We also assume that local variables stay local and shared variables and locks stay shared throughout the execution of the program. These restrictions are done to simplify the presentation and they are not inherent to our approach.

Formally, we define the set of variables for a concurrent program with n threads as:

$$\mathbf{Var} = \mathbf{ShaVar} \cup \mathbf{LockVar} \cup \bigcup_{t \in [1..n]} \mathbf{TLocVar}_t$$

where **ShaVar** (shared), **LockVar** (lock), and $\mathbf{TLocVar}_t$ for $t \in [1..n]$ (local variables) are all disjoint sets.

In order to formulate the program states and executions, we first define the program stores in a concurrent program. A concurrent program consists of a shared store $S \in \mathbf{Sha} = \mathbf{ShaVar} \rightarrow \mathbf{Dom}$, a lock store $L \in \mathbf{Lock} = \mathbf{LockVar} \rightarrow \mathbf{Dom}$, and a local store for each thread t, $T_t \in \mathbf{TLoc}_t = \mathbf{TLocVar}_t \rightarrow \mathbf{Dom}$. To simplify our presentation we use **Dom** to denote the union of all variable domains and assume that each variable is mapped to a type correct value. Note that, the program counter of a thread t is represented as a local variable in $\mathbf{TLocVar}_t$.

We define the set of states for a concurrent program with n threads as: $\mathbf{State} = \mathbf{Sha} \times \mathbf{Lock} \times \mathbf{W} \times \mathbf{R} \times \mathbf{TLoc}_1 \times \mathbf{TLoc}_2 \times \ldots \times \mathbf{TLoc}_n$. In addition to previous stores, a state $s \in \mathbf{State}$ also consists of a wait store $W \in \mathbf{W}$ that keeps track of the blocked threads and the operations that they blocked on, and a reentry store $R \in \mathbf{R}$ that keeps track of the reentry count for each thread and each lock. We formally define wait and reentry stores when we discuss lock operations below.

We denote the initial states of a concurrent program as $\mathbf{Init} \subseteq \mathbf{State}$. Given a state of a concurrent program $s = (S, L, W, R, T_1, T_2, \ldots, T_n) \in \mathbf{State}$ and a shared variable $v \in \mathbf{ShaVar}$, $S(v)$ denotes the value of the variable v in state s. The values of lock and local variables in state s are similarly defined by the lock store (L) and the local stores for the threads (T_t), respectively.

We define the set of operations of a concurrent program as: $\mathbf{Op} = \mathbf{SOp} \cup \mathbf{LOp} \cup \mathbf{TOp}_1 \cup \mathbf{TOp}_2 \cup \ldots \cup \mathbf{TOp}_n$ where **SOp** denotes the shared operations, **LOp** denotes the lock operations, and $\mathbf{TOp}_t$ denotes the local operations for thread t. Each operation corresponds to an atomic step of execution, i.e., given a program state, execution of an operation causes the program to transit to another program state. The operations are formally defined as:

$$\begin{array}{lcl} \mathbf{SOp} & = & \mathbf{Sha} \rightarrow \mathbf{Sha} \\ \mathbf{LOp} & = & \mathbf{Lock} \times \mathbf{W} \times \mathbf{R} \rightarrow \mathbf{Lock} \times \mathbf{W} \times \mathbf{R} \\ \mathbf{TOp}_t & = & \mathbf{TLoc}_t \rightarrow \mathbf{TLoc}_t \text{ for } t \in [1..n] \end{array}$$

Based on operations, we define the transition relation of a concurrent program as follows $\mathbf{Trans} \subseteq \mathbf{State} \times \mathbf{Op} \times \{1..n\} \times \mathbf{State}$, i.e., we label a transition with the corresponding operation and the index of the thread that executes the operation. Given a transition $(s, o, t, s') \in \mathbf{Trans}$ where $s = (S, L, W, R, T_1, T_2, \ldots, T_n)$ and $s' =$

$(S', L', W', R', T'_1, T'_2, \ldots, T'_n)$, we have the following:

$$
\begin{array}{lcl}
o \in \mathbf{SOp} & \Rightarrow & S' = o(S) \wedge L' = L \wedge \bigwedge_{i \in [1..n]} T'_i = T_i \\
o \in \mathbf{LOp} & \Rightarrow & S' = S \wedge (L', W', R') = o(L, W, R) \wedge \\
 & & \bigwedge_{i \in [1..n]} T'_i = T_i \\
o \in \mathbf{TOp}_t & \Rightarrow & S' = S \wedge L' = L \wedge T'_t = o(T_t) \wedge \\
 & & \bigwedge_{i \in [1..n], i \neq t} T'_i = T_i
\end{array}
$$

i.e., a shared operation can only change the values of the shared variables, a lock operation can only change the values of the lock variables, the wait store and the reentry store, and a local operation of a thread can only change the values of that thread's local variables.

We use $s \xrightarrow{o,t} s'$ as an alternative notation for the tuple (s, o, t, s'). We use an interleaving semantics, and define a run ρ of a concurrent program as: $\rho = s_0 \xrightarrow{o_0,t_0} s_1 \xrightarrow{o_1,t_1} s_2 \ldots$ where $s_0 \in$ **Init** and for all i, $s_i \xrightarrow{o_i,t_i} s_{i+1} \in$ **Trans**.

Lock Operations: We assume that lock operations are implemented as *guarded commands*, i.e., each lock operation $o \in \mathbf{LOp}$ has a guard $o.g$ and an update $o.u$ where $o.g$: **Lock** $\rightarrow \{\text{TRUE}, \text{FALSE}\}$, $o.u$: **Lock** $\rightarrow$ **Lock**. The update of a lock operation ($o.u$) is only executed when its guard ($o.g$) evaluates to true.

In order to keep track of the threads that are waiting on a blocked lock operation we use the wait store $W \in \mathbf{W}$ that maps each thread to either a lock operation or $\perp$, $W \in \mathbf{W}$: $\{1 \ldots n\} \rightarrow \mathbf{LOp} \cup \{\perp\}$ where $W(t) = \perp$ means that the thread t is not blocked (i.e., it is not waiting), and $W(t) = o$ means that the thread t was blocked while executing the operation o and it is waiting for the guard of the operation o to become true. In our execution model $W(t) = o$ implies that 1) the last operation executed by thread t was o which caused thread t to block, and 2) the next operation that will be executed by t will also be o, however, the next execution of o will not block.

Since our goal is to model reentrant locks, once a lock acquire operation is completed by a thread (i.e., the thread is not blocked and waiting), the next execution of the same lock acquire operation by the same thread should not block unless the lock is released. In order to model this semantics, we partition the lock operations to *acquire* and *release* operations where $\mathbf{LOp} = \mathbf{LAOp} \cup \mathbf{LROp}$. We further assume that each lock operation is associated with a unique lock. Assuming that there are m locks, then the lock operations are partitioned to m sets of acquire operations and m sets of release operations where $\mathbf{LOp} = \mathbf{LAOp}_1 \cup \mathbf{LAOp}_2 \cup \ldots \cup \mathbf{LAOp}_m \cup \mathbf{LROp}_1 \cup \mathbf{LROp}_2 \cup \ldots \cup \mathbf{LROp}_m$. This does not imply that there are m separate sets of lock variables. For example, in our terminology a read-write lock will correspond to one read lock and one write lock whereas the operations implementing these two locks will be using the same set of lock variables. The reentrant condition is expressed as follows: An acquire operation $o \in \mathbf{LAOp}_i$ executed by thread t should not block if the number of operations from $\mathbf{LAOp}_i$ that have been executed by thread t are greater than the number of operations from $\mathbf{LROp}_i$ that have been executed by thread t. We formalize this by using a reentry store $R \in \mathbf{R}$ that maps each thread and each lock to the difference between the number of executions of the acquire and release operations: $R \in \mathbf{R}$: $\{1 \ldots n\} \rightarrow \{1 \ldots m\} \rightarrow$ **Int** where $R(t)(i)$ denotes the difference between the number of executions of the acquire and release operations of lock i by thread t. Figure 3 summarizes the semantics of lock operations.

III. Lock Interfaces and Interface Correctness

In this section, we first formalize the lock interfaces. A lock interface specifies correct ordering of lock and shared operations for each thread. We define the interface correctness condition for threads based on lock interfaces and show that the interface correctness of each thread can be checked individually. As we discuss in the following section, the interface correctness property, i.e., the property that establishes that all threads obey the lock interface, enables us to use non-reentrant lock behaviors to simulate reentrant lock behaviors in a concurrent system. At the end of this section we also discuss how we use Java Path Finder (JPF) to check the interface correctness property.

Given a run $\rho = s_0 \xrightarrow{o_0,t_0} s_1 \xrightarrow{o_1,t_1} s_2 \ldots$, $\pi(\rho)(t) \in (\mathbf{SOp} \cup \mathbf{LOp})^*$ denotes the shared-trace of run ρ for thread t. We define the projection function π below.

Let ρ_i denote the ith prefix of ρ (which consists of $i+1$ states and i transitions). We denote the $i+1$st prefix of ρ as $\rho_{i+1} = \rho_i \xrightarrow{o_i,t_i} s_{i+1}$. We define π inductively as follows, $\pi(\rho_0)(t) = \epsilon$, and

- $\pi(\rho_{i+1})(t) = \pi(\rho_i)(t)\ o_i$ if $t_i = t$ and either $o_i \in \mathbf{SOp}$ or $o_i \in \mathbf{LOp}$ and $W_{i+1}(t) = \perp$ where W_{i+1} is the wait store for state s_{i+1},
- $\pi(\rho_{i+1})(t) = \pi(\rho_i)(t)$ otherwise.

We call $\pi(\rho)(t)$ a *lock word*. A *lock interface* is a language $\mathbf{LI} \subseteq (\mathbf{SOp} \cup \mathbf{LOp})^*$. A concurrent program is interface correct if for each run ρ and for each thread t, $\pi(\rho)(t) \in \mathbf{LI}$.

Lock Interface Machine: A *lock interface machine* is an extended finite state machine (EFSM) that recognizes the lock interface $\mathbf{LI} \subseteq (\mathbf{SOp} \cup \mathbf{LOp})^*$. A thread is interface correct if each lock word generated by that thread is accepted by the lock interface machine. In other words, lock interface machine defines the acceptable sequences of lock and shared operations that can be generated by a single thread.

Formally, given a concurrent program P with m locks, the lock interface machine $\mathbf{M^I}$ is a tuple $(S, S_0, C, C_0, \Sigma, \delta, F)$, where:

- $S = 2^{\{lock_1, .., lock_m\}}$ is a finite set of states that is defined by the powerset of the m locks where $lock_i \in s$ means that the lock i is held by the thread.
- $S_0 \subseteq S$ is the set of initial states and is defined as $\{\emptyset\}$.
- $C = \{1 \ldots m\} \rightarrow$ **Int** is the reentry store for a single thread denoting the reentry-counts for that thread. Each

Given a lock operation $o \in \mathbf{LOp}$, a thread t and two states $s = (S, L, W, R, T_1, T_2, \ldots, T_n)$ and $s' = (S, L', W', R', T_1, T_2, \ldots, T_n)$ where $(L', W', R') = o(L, W, R)$, then $s \xrightarrow{o,t} s'$ iff one of the following cases hold:

Case: Lock acquire call ($o \in \mathbf{LAOp}_k$), thread does not hold the lock ($R(t)(k) = 0$), guard does not hold ($o.g(L) = \text{FALSE}$):
$o \in \mathbf{LAOp}_k \wedge R(t)(k) = 0 \wedge o.g(L) = \text{FALSE} \Rightarrow$
- Thread t starts waiting: $W(t) = \bot$, $W'(t) = o$, for all $i \in [1..n]$, $i \neq t \Rightarrow W'(i) = W(i)$
- Lock and reentry stores do not change: $L' = L$, $R' = R$

Case: Lock acquire call ($o \in \mathbf{LAOp}_k$), thread does not hold the lock ($R(t)(k) = 0$), guard holds ($o.g(L) = \text{TRUE}$):
$o \in \mathbf{LAOp}_k \wedge R(t)(k) = 0 \wedge o.g(L) = \text{TRUE} \Rightarrow$
- Thread t does not wait: $W(t) = o \vee W(t) = \bot$, $W'(t) = \bot$, for all $i \in [1..n]$, $i \neq t \Rightarrow W'(i) = W(i)$
- Lock state is updated: $L' = o.u(L)$
- Reentry count for t is set to 1: $R'(t, k) = 1$, for all $i \in [1..n]$, $j \in [1..m]$, $(i \neq t \vee j \neq k) \Rightarrow R'(i)(j) = R(i)(j)$

Case: Lock acquire call ($o \in \mathbf{LAOp}_k$), thread holds the lock ($R(t)(k) \geq 1$):
$o \in \mathbf{LAOp}_k \wedge R(t)(k) \geq 1 \Rightarrow$
- Reentry count for t is incremented: $R'(t)(k) = R(t)(k) + 1$, for all $i \in [1..n]$, $j \in [1..m]$, $(i \neq t \vee j \neq k) \Rightarrow R'(i)(j)$
- Lock and wait stores do not change: $L' = L$, $W' = W$

Case: Lock release call ($o \in \mathbf{LROp}_k$), thread is ready to release the lock ($R(t)(k) = 1$), guard does not hold ($o.g(L) = \text{FALSE}$):
$o \in \mathbf{LROp}_k \wedge R(t)(k) = 1 \wedge o.g(L) = \text{FALSE} \Rightarrow$
- Thread t starts waiting: $W(t) = \bot$, $W'(t) = o$, for all $i \in [1..n]$, $i \neq t \Rightarrow W'(i) = W(i)$
- Lock and reentry stores do not change: $L' = L$, $R' = R$

Case: Lock release call ($o \in \mathbf{LROp}_k$), thread is ready to release the lock ($R(t)(k) = 1$), guard holds ($o.g(L) = \text{TRUE}$):
$o \in \mathbf{LROp}_k \wedge R(t)(k) = 1 \wedge o.g(L) = \text{TRUE} \Rightarrow$
- Thread t does not wait: $W(t) = o \vee W(t) = \bot$, $W'(t) = \bot$, For all $i \in [1..n]$, $i \neq t \Rightarrow W'(i) = W(i)$
- Lock state is updated: $L' = o.u(L)$
- Reentry count for t is set to 0: $R'(t, k) = 0$, for all $i \in [1..n]$, $j \in [1..m]$, $(i \neq t \vee j \neq k) \Rightarrow R'(i)(j) = R(i)(j)$

Case: Lock release call ($o \in \mathbf{LROp}_k$), thread is not ready to release the lock ($R(t)(k) > 1$):
$o \in \mathbf{LROp}_k \wedge R(t)(k) > 1 \Rightarrow$
- Reentry count for t is decremented: $R'(t)(k) = R(t)(k) - 1$, for all $i \in [1..n]$, $j \in [1..m]$, $(i \neq t \vee j \neq k) \Rightarrow R'(i)(j)$
- Lock and wait stores do not change: $L' = L$, $W' = W$

Figure 3. Semantics of lock operations

$c \in C$ is an integer vector and $c(i)$ denotes the reentry-count for lock i.

- $C_0 \subseteq C$ is the set of initial values for the reentry-counts where $C_0 = \{c \mid \forall i \in [1..m], c(i) = 0\}$.
- Σ is the finite alphabet defined as $\Sigma = \mathbf{SOp} \cup \mathbf{LOp}$.
- $\delta \subseteq S \times C \times \Sigma \times S \times C$ is the transition relation where $(s, c, o, s', c') \in \delta$ if and only if one of the following three conditions holds:
  - $o \in \mathbf{LAOp}_i \wedge s' = s \cup \{lock_i\} \wedge c'(i) = c(i) + 1$,
  - $o \in \mathbf{SOp} \wedge s' = s \wedge c' = c$,
  - $o \in \mathbf{LROp}_i \wedge lock_i \in s \wedge ((c(i) = 1 \wedge s' = s \setminus \{lock_i\} \wedge c'(i) = 0) \vee (c(i) > 1 \wedge s' = s \wedge c'(i) = c(i) - 1))$.
- $F \subseteq S$ is the set of final states defined as $F = \{\emptyset\}$.

Consider a read-write lock as an example. Let $locks = \{r, w\}$, $\mathbf{SOp} = \{\texttt{read}, \texttt{write}\}$, $\mathbf{LOp} = \mathbf{LAOp}_r \cup \mathbf{LAOp}_w \cup \mathbf{LROp}_r \cup \mathbf{LROp}_w$, where $\mathbf{LAOp}_r = \{\texttt{read\_enter}\}$, $\mathbf{LAOp}_w = \{\texttt{write\_enter}\}$, $\mathbf{LROp}_r = \{\texttt{read\_exit}\}$, and $\mathbf{LROp}_w = \{\texttt{write\_exit}\}$.

A read-write lock interface machine is defined as $\mathbf{M^I} = (S, S_0, C, C_0, \Sigma, \delta, F)$, where: $S = \{\emptyset, \{r\}, \{w\}, \{r, w\}\}$, $S_0 = \{\emptyset\}$, $C = \{r, w\} \rightarrow \mathbf{Int}$, $C_0 = \{c \mid c(r) = 0 \wedge c(w) = 0\}$, $\Sigma = \{\texttt{read}, \texttt{write}, \texttt{read\_enter}, \texttt{write\_enter}, \texttt{read\_exit}, \texttt{write\_exit}\}$, $F = \{\emptyset\}$ and δ consists of the following transitions

- $(s, c, \texttt{read}, s, c)$, where $s \in \{\{r\}, \{w\}, \{r, w\}\}$.
- $(s, c, \texttt{write}, s, c)$, where $s \in \{\{w\}, \{r, w\}\}$.
- $(s, c, \texttt{read\_enter}, s', c')$, where $s' = s \cup \{r\}$ and $c'(r) = c(r) + 1$.
- $(s, \texttt{read\_exit}, s')$, where $s \in \{\{r\}, \{r, w\}\}$, and $(c(r) = 1 \wedge s' = s \setminus \{r\} \wedge c'(r) = 0) \vee (c(r) > 1 \wedge s' = s \wedge c'(r) = c(r) - 1)$.
- $(s, c, \texttt{write\_enter}, s', c')$, where $s' = s \cup \{w\}$ and $c'(w) = c(w) + 1$.
- $(s, c, \texttt{write\_exit}, c', s')$, where $s \in \{\{w\}, \{r, w\}\}$, and $(c(w) = 1 \wedge s' = s \setminus \{w\} \wedge c'(w) = 0) \vee (c(w) > 1 \Rightarrow s' = s \wedge c'(w) = c(w) - 1)$.

A lock run over a lock word $o_1 o_2 \ldots o_k \in (\mathbf{SOp} \cup \mathbf{LOp})^*$ is a finite sequence $(s_0, c_0)(s_1, c_1)(s_2, c_2) \ldots (s_k, c_k)$ with $(s_0, c_0) \in S_0 \times C_0$, such that $\forall i \in [1..k], (s_i, c_i, o_i, s_{i+1}, c_{i+1}) \in \delta$. A lock run is accepting if $s_k \in F$ and $\forall i \in [1..m], c_k(i) = 0$. A lock word w is accepted by $\mathbf{M^I}$ if there exists an accepting lock run over w. Then, the lock interface $\mathbf{LI}$ for thread t is the set of lock words that $\mathbf{M^I}$ accepts, denoted as $\mathbf{LI} = L(\mathbf{M^I})$.

Definition 3.1: A thread t in a concurrent program is *interface correct* if for each run ρ, the lock word $\pi(\rho)(t) \in \mathbf{LI} = L(\mathbf{M^I})$. A concurrent program is interface correct if all the threads in the program are interface correct.

Interface Verification: During interface verification we verify each thread in the program separately using the Java Path Finder (JPF) [21]. If a thread invokes the lock and shared operations in an order that is not accepted by the lock interface machine, then the thread is not interface correct.

Our interface verification approach is thread modular, i.e., we check each thread separately for interface violations. For this purpose, we isolate each thread by a conservative approximation of the behavior of other threads in the distributed program without modifying the thread code. We achieve this by replacing the lock and shared operations

with stubs. The lock operation stubs keep track of the lock interface machine state and the reentry count for each lock. Although the lock variables are abstracted away, since we are checking reentrant locks we have to keep track of the reentry counts in the lock operation stubs. The return values for the shard operation stubs are chosen non-deterministically to model arbitrary manipulation of the shared data by the other threads. Note that during interface verification, JPF searches the state space for all possible outcomes of the non-deterministic choices that are inserted during thread isolation. The thread isolation techniques we use are explained in detail in [5].

The lock and shared operation stubs have assertions that check the ordering constraints based on the lock interface machine. If JPF reports a violation of an assertion in a lock or shared operation stub, then we know that the thread in question caused an interface violation. JPF outputs a counter-example execution trace that leads to the violation of the assertion.

IV. Lock Behavior and Behavior Verification

In this section, we present an abstract model for lock behaviors of a concurrent system. We argue that this abstract model can simulate the behavior of the concurrent system whose threads behave according to the lock interface specifications.

Lock Behavior Machine: A *lock behavior machine* is a special type of EFSM which contains one state machine for each thread, i.e., it is a product machine characterizing the combined behavior of all the threads. Formally, given a concurrent program P with n threads, a lock behavior machine $\mathbf{M^B}$ is a tuple $(C, C_0, \Sigma, \mathbf{M}_1, \mathbf{M}_2, \ldots, \mathbf{M}_n)$, where:

- $C = \textbf{Lock} = \textbf{LockVar} \rightarrow \textbf{Dom}$ denotes the set of lock stores.
- C_0 is the set of initial lock stores.
- Σ is the finite alphabet defined as $\Sigma = \textbf{LOp} \cup \{\tau\}$.
- For $t \in [1..n], \mathbf{M}_t = (S_t, S_{t0}, \delta_t, F_t)$, where
  - $S_t = 2^{\{lock_1,..,lock_m\}}$ is the set of states that is defined by the powerset of the m locks where $lock_i \in s$ means that the lock i is held by the thread t.
  - $S_{t_0} \subseteq S_t$ is the set of initial states and is defined as $\{\emptyset\}$.
  - $\delta_t \subseteq S_t \times C \times \Sigma \times S_t \times C$ is the transition relation where $(s_t, c, o, s'_t, c') \in \delta_t$ if and only if one of the following three conditions holds:
    - $o \in \textbf{LAOp}_i \wedge lock_i \notin s_t \wedge o.g(c) = \text{TRUE} \wedge s'_t = s_t \cup \{lock_i\} \wedge c' = o.u(c)$,
    - $o = \tau \wedge s'_t = s_t \wedge c' = c$,
    - $o \in \textbf{LROp}_i \wedge lock_i \in s_t \wedge o.g(c) = \text{TRUE} \wedge s'_t = s_t \setminus \{lock_i\} \wedge c' = o.u(c)$.
  - $F_t \subseteq S_t$ is the set of final states defined as $F_t = \{\emptyset\}$.

Consider the read-write lock example. We define the read-write lock behavior machine using two lock variables where $busy$ denotes that a writer is in the critical section and nR denotes the number of reader threads in the critical section. A read-write lock behavior machine with n threads can be defined as: $\textbf{LockVar} = \{busy, nR\}, \textbf{Dom}(busy) = \{\text{FALSE}, \text{TRUE}\}, \textbf{Dom}(nR) = \textbf{Int}, C = \textbf{LockVar} \rightarrow \textbf{Dom}$, $C_0 = \{c \mid c \in C, c(busy) = \text{FALSE} \wedge c(nR) = 0)\}$, $\Sigma = \{\tau, \texttt{read\_enter}, \texttt{write\_enter}, \texttt{read\_exit}, \texttt{write\_exit}\}$. For each $\mathbf{M}_t = (S_t, S_{t0}, \delta_t, F_t)$, $S_t = \{\emptyset, \{r\}, \{w\}, \{r, w\}\}$, $S_{t_0} = \{\emptyset\}$, $F = \{\emptyset\}$ and δ_t consists of the following transitions

- (s_t, c, τ, s_t, c),
- $(s_t, c, \texttt{read\_enter}, s'_t, c')$, where $c(busy) = \text{FALSE}, s'_t = s_t \cup \{r\}$, $c'(busy) = c(busy), c'(nR) = c(nR) + 1$.
- $(s_t, \texttt{read\_exit}, s'_t)$, where $s'_t = s_t \setminus \{r\}, c'(busy) = c(busy)$, $c'(nR) = c(nR) - 1$.
- $(s_t, c, \texttt{write\_enter}, s'_t, c')$, $c(busy) = \text{FALSE}, c(nR) = 0, s'_t = s_t \cup \{w\}, c'(busy) = \text{TRUE}, c'(nR) = c(nR)$.
- $(s_t, c, \texttt{write\_exit}, s'_t, c')$, where $s'_t = s_t - \{w\}, c'(busy) = \text{FALSE}$, $c'(nR) = c(nR))$.

The semantics of a lock behavior machine is defined as a transition system $T = (S, S_0, \Delta)$, where

- $S = C \times S_1 \times \ldots \times S_n$.
- $S_0 = C_0 \times S_{1_0} \times \ldots \times S_{n_0}$.
- $\Delta = \bigcup_{t \in [1..n]} \{\delta \wedge \bigwedge_{i \neq t} (s'_i = s_i) \mid \delta \in \delta_t\}$.

Note that, the lock behavior machine does not contain a reentry store that keeps track of reentry counts. As far as the lock behavior verification is concerned, the reentry counts can be abstracted away. We formalize this by defining a simulation relation between a concurrent program and its lock behavior machine. Consider a concurrent program $P = (\textbf{State}, \textbf{Init}, \textbf{Trans})$ with n threads and a lock behavior machine $\mathbf{M^B} = (C, C_0, \Sigma, \mathbf{M}_1, \mathbf{M}_2, \ldots, \mathbf{M}_n)$ that defines the behaviors of the locks used by P. We will define a simulation relation H between P and $\mathbf{M^B}$ as follows: $H \subseteq \textbf{State} \times (C \times S_1 \times \ldots \times S_n)$ where $((S, L, W, R, T_1, T_2, \ldots, T_n), (c, s_1, \ldots, s_n)) \in H$, if and only if,

- for all $v \in \textbf{LockVar}, L(v) = c(v)$,
- for all $t \in [1..n]$ and for all $j \in [1..m]$, $R(i)(j) \geq 1 \Leftrightarrow lock_j \in s_i$.

During lock behavior verification we restrict the set of atomic properties to two types of properties: 1) The properties on lock variables. 2) Properties in the form of "thread t holds lock $lock_i$." Based on this set of atomic properties and using the simulation relation above, we can show the following result:

Theorem 4.1: Given a program P, if all the threads in P are interface correct, then the lock behavior machine $\mathbf{M^B}$ simulates P, and if an ACTL property holds for $\mathbf{M^B}$, then it holds for P.

This property can be proven by first showing that the simulation relation H preserves the atomic properties defined above. Given $((S, L, W, R, T_1, T_2, \ldots, T_n), (c, s_1, \ldots, s_n)) \in H$ the atomic properties on lock variables are preserved since the lock variables have the same values by definition, as defined by L and c. The property "thread t holds lock $lock_i$" is equivalent to the property $R(t)(i) \geq 1$ and $lock_i \in s_t$. And again this property is preserved by the definition of the simulation relation.

Next, we need to show that for any transition in the concurrent program there exists a matching transition in the lock behavior machine. This can be proved using the following observation. Only transitions in the concurrent program that change the values of the lock variables are those in which a reentry-count change from 0 to 1 or vice versa. These transitions can be mapped to the corresponding transitions of the lock behavior machine. All other transitions are mapped to τ transitions since they do not change the values of the lock variables or the ownership of the locks.

This property implies that we can verify properties about lock variables in a modular fashion. Given an ACTL property, we first verify that all the threads in a program are interface correct. Next, we check the given property on the lock behavior machine. If the property holds for the lock behavior machine and if all the threads are interface correct, then we conclude that the concurrent program satisfies the given property.

Behavior Verification: We use the Action Language Verifier (ALV) [22] for behavior verification. An Action Language specification consists of integer, boolean and enumerated variables, parameterized integer constants, and a set of modules and actions which are composed using synchronous and asynchronous composition operators.

We map the lock behavior machine to an Action Language specification by converting guarded commands of the lock implementation to actions in the Action Language specification. We use the template for guarded commands from [3]. If a lock is implemented in Java using this template, our translator automatically constructs the lock behavior machine in Action Language.

ALV is an infinite state model checker which verifies CTL properties of Action Language specifications with unbounded integer variables (such as `nR`). For the infinite state systems that can be specified in Action Language, model checking is undecidable. Hence, ALV uses conservative approximations techniques during verification. The undecidability of the model checking problem for ALV implies that the fixpoint computations are not guaranteed to converge. ALV uses several conservative approximation heuristics to achieve convergence. For the experiments we conducted in this study ALV was able to verify all the lock implementations. I.e., all the fixpoint computations converged and the approximations were precise enough to verify the given properties.

Since ALV allows unbounded integer variables, it enables us to use an automated abstraction technique, called counting abstraction, to verify the lock implementations with respect to arbitrary number of threads [3], [23]. The counting abstraction technique [8] in ALV supports verification of parameterized systems with an arbitrary number of finite state modules. The basic idea is to define an abstract transition system in which the local states of the threads (corresponding to the local states of the threads in the lock behavior machine) are abstracted away, but the number of threads in each local state is counted by introducing a new integer variable for each interface state. For example, for the read-write lock we will have one integer variable counting the number of threads in the $\emptyset$ state, one for counting the threads in the $\{r\}$ state, etc. The initial states and the transition relation of the parameterized system is defined using linear arithmetic constraints on these variables. A parameterized integer constant, `n`, denotes the number of threads. This parameterized constant is restricted to be positive and when the specification is verified with ALV, the results hold for any valuation of this parameterized constant (i.e. the results are valid for any number of threads) for ACTL properties.

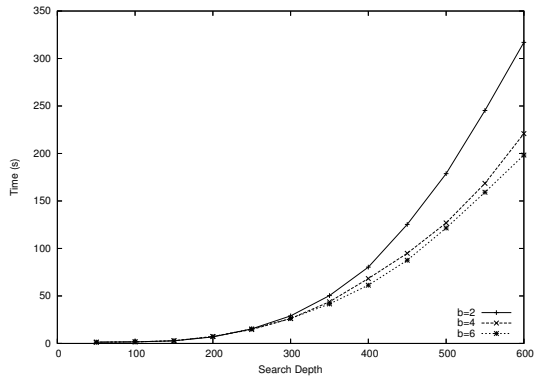
V. Experiments

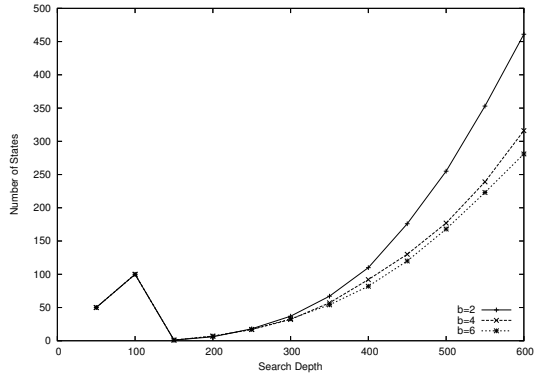
Experiments with a concurrent buffer: For large software systems, a non-modular verification approach is simply not feasible. So we are unable to compare our modular verification approach with a non-modular approach on a large software system. Experimenting on a small program enables us to compare the verification performance of our modular approach with a non-modular approach.

We implemented a Java program that consists of a concurrent buffer that stores an array of integers that are accessed by two threads. The producer thread inserts new values to the buffer and the consumer thread removes values from the buffer. The buffer is protected by a reentrant read-write lock. Both threads can peek at the buffer before modifying it by obtaining the read lock. A thread has to obtain the write lock before inserting or removing an item from the buffer.

We checked for interface violations for this program. For example, if a thread calls the peek method of the concurrent buffer before calling the read-enter method of the read-write lock, interface verification should report an interface violation. Since the read-write lock is reentrant interface violation should take reentry counts into account. For example, if a thread calls read-enter twice then read-exit twice and then peek, there is an interface violation. However, if a thread calls read-enter twice then read-exit once and then peek, there is no interface violation.

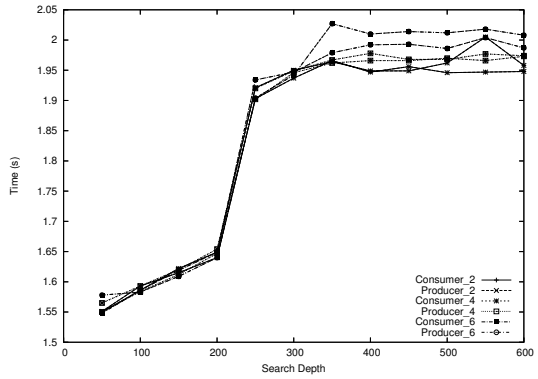
In our experiments we experimented with different levels of nesting between the lock entry and exit operations. We created three versions of the program where the reentry depths were bounded by 2, 4 or 6.

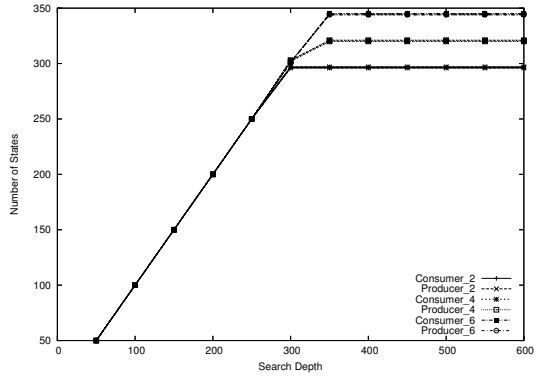




(in the graph above the y-axis shows thousands of states)

(a)





(b)

Figure 4. Verification time and number of states explored for (a) non-modular and (b) modular interface verification

During non-modular verification we used JPF to look for interface violations on the whole program. During modular verification we first isolated the producer and consumer threads by replacing the lock class with the lock stub (i.e., the lock interface machine) and the concurrent buffer with a buffer stub. We checked for interface violations in the producer and consumer threads separately. When we checked for interface violations in the whole program using JPF (i.e., the non-modular approach), JPF ran out of memory for all configurations. Therefore, we bounded the search depth of JPF. We limited the maximum search depth for JPF to values between 50 and 600. JPF was able to explore the whole state space using the modular verification approach for all configurations.

The results of our experiments are shown in Figure 4. The exponential increase in the number of reachable states and the verification time with respect to increasing search depth is clearly visible in the non-modular interface verification. This is not a surprising since it is the result of the well-known state-explosion problem. When the search depth is 600 and the reentry depth is limited to 6, the number of states explored by JPF is 461 thousand and JPF takes 316 seconds to explore all the states within that depth. However, for the modular behavior verification, the situation is quite different. JPF is able to explore the full-state space of each thread with a search depth of 350 states and the verification time is around 2 seconds. For this simple example, it is clear that the modular verification approach is significantly more efficient than a non-modular verification strategy.

Experiments with TSAFE: In order to investigate the effectiveness of the proposed approach on a real application, we conducted experiments on a software component called the Tactical Separation Assisted Flight Environment (TSAFE). TSAFE is part of a framework developed by NASA researchers which targets the automation of the air traffic control system [10]. TSAFE is an implementation of a safety critical component in this framework [9]. TSAFE performs the following functions: 1) Display aircraft position 2) Display aircraft planned route 3) Display aircraft future projected route trajectory 4) Indicate conformance problems.

The TSAFE implementation consists of 21,057 lines of Java source code in 87 classes. A server component stores the trajectories of the flights in a flight database. The feed parser thread in the server receives updates of the locations of the flights periodically from the radar feed through a network connection and updates the trajectory database. A computation component in the server implements the trajectory synthesis and conformance monitoring functions. The client side implements the display functionality in a GUI. Multiple clients can connect to the server at the same time via RMI. A timer thread at the server periodically prompts the clients to access the flight database to obtain the current data.

In [4], [5] TSAFE implementation was verified with re-

Table I
VERIFICATION PERFORMANCE

Thread	Time(s)		Memory (MB)	
	ave	std	ave	std
ServerEvent	11.62	0.14	22.14	2.39
ServerRMI	177.85	10.24	63.15	13.13
ServerFeed	194.78	7.51	52.44	13.14
ClientEvent	451.54	0.63	82.55	4.58
ClientRMI	11.82	1.54	26.54	3.40

spect to non-reentrant read-write and mutex locks. We modified the TSAFE implementation using reentrant read-write and mutex locks. Reentrant locks allow a more fine-grained locking mechanism that is less likely to block threads, potentially increasing the performance of the system. For example, in the earlier version of TSAFE the lock that protects the flight database is grabbed at a high level which leads to course grain synchronization. Using a reentrant lock it is possible to achieve a fine-grain synchronization where each operation grabs the lock separately.

Behavior Verification: TSAFE implementation uses two instances of a read-write lock and three instances of a mutex lock. This means that for behavior verification we only need to check two lock implementations. The lock behavior specifications for these two lock implementations are identical to the ones analyzed in [4], [5]. Although, the locks in [4], [5] are not reentrant, the modular verification approach we present in this paper reduces the behavior verification problem for reentrant locks to behavior verification problem for non-reentrant locks. Hence the results reported in [4], [5] also hold for reentrant lock verification. We report them here for completeness. We identified 10 ACTL properties for the read-write lock (such as AG(`busy` $\Rightarrow$ `nR`$= 0$)) and 6 ACTL properties for the mutex lock. Using an Action Language specification that corresponds to a lock behavior machine that contains 16 threads, the verification of the read-write lock takes 3.42 seconds and uses 9.36 MBytes of memory and the verification of the mutex lock takes 0.02 seconds and uses 0.62 MBytes of memory.

We also checked the parameterized version of the lock behavior machines using counting abstraction. We identified 6 ACTL properties for the parameterized read-write lock specification (such as AG($\neg(count_{\{w\}} > 1)$) where $count_{\{w\}}$ denotes the number of threads that hold the write lock) and 4 ACTL properties for the parameterized mutex lock specification. Using an Action Language specification the verification of the parameterized read-write lock specification takes 0.21 seconds and uses 12.05 MBytes of memory and the verification of the parameterized mutex lock specification takes 0.03 seconds and uses 0.24 MBytes of memory.

Interface Verification: Table I shows the verification performance for interface verification for the five threads in TSAFE using the JPF model checker. In order to do interface verification, in addition to isolating the threads from each other, we also have to model the environment of the program, i.e., generate the input events. We used the approach used in [5] to generate drivers that provide non-deterministically generated input event sequences. Note that, during verification JPF model checker tries all possible outcomes for the non-deterministic choices. In order to achieve convergence during interface verification, we bound the length of the input event sequences which means that this verification approach is not sound. Combining interface checking with automated abstraction techniques such as predicate abstraction can be helpful for analyzing threads with large state spaces and achieving sound verification.

In order to investigate the bug finding capability of our framework during interface verification, we automatically injected random faults into the thread implementations. Then using our framework we tried to find the interface violations caused by these faults.

We categorize the automatically generated interface faults using the classification from [4], [5] 1) *modified-call faults* (IM) which were generated by removing, adding or swapping calls to the methods of the locks, and 2) *conditional-call faults* which were generated by adding a branch condition in front of a method call to a lock method. The conditional-call faults are further categorized as: a) *program-variable faults* (ICV) in which the created branch conditions used existing program variables, and b) *new-variable faults* (ICN) in which the created branch conditions used new variables that were declared during fault creation.

The automated fault injection program works as follows. The IM faults are injected by adding, removing or swapping lock methods before or after a shared data access by the program. To introduce an ICV fault, the program chooses an integer or boolean class field and inserts a conditional using this field before a lock method call statement. The ICN faults are injected by creating a new unique integer variable, and adding a conditional statement using this new variable before a method call to a lock method. This new variable is initialized to zero and is incremented every time the control reaches the inserted conditional statement. The conditional is of the form `if (_new_var00 < C)` where `C` is a constant integer value that can be 50, 60, 70, 80, 90, 100, 200, 300, 400, or 500.

We generated 20 faulty versions of TSAFE (denoted as v1-20) by fault injection and the faults are categorized as follows IM faults {v10, v11, v12, v15, v18, v19}; ICV faults {v8, v13, v17}; and ICN faults {v3, v4, v8, v12, v13, v17, v19, v20}.

The results reported in Table II show that our approach is effective in catching interface violations. Some of the faults were spurious, i.e., they are changes in the program which do not cause interface violations. All the faults that were not spurious were caught by JPF during interface verification. I.e., they caused assertion violations either in the lock or share data operation stubs.

Table II
FALSIFICATION PERFORMANCE

ServerEvent	v8:	(ICV) Not found, spurious fault
	v12:	(ICN) Found in 8.23 sec and 53.29MB
	v17:	(ICN) Found in 14.56 sec and 125.85MB
ServerRMI	v3 :	(ICN) Found in 33.8s and 343.13MB
	v8:	(ICV) Not found, spurious fault
	v10:	(IM) Found in 31.27 sec and 329.1MB
	v11:	(IM) Found in 30.99 sec and 318.45MB
	v12:	(ICN) Found in 38.38 sec and 403.18MB
	v17:	(ICV) Not found, spurious fault
	v18:	(IM) Found in 31.40 sec and 322.48MB
ServerFeed	v4:	(ICN) Found in 122.59 sec and 996.82MB
	v8:	(ICN) Found in 114.59 sec and 831,35 MB
	v17:	(ICN) Found in 15.15 sec and 134.54MB
	v19:	(IM) Found in 2.78 sec and 7MB; (ICN) Not found, spurious fault
ClientEvent		No faults, none found
ClientRMI	v12:	(IM) Found in 6.21 sec and 40.59MB
	v13:	(ICN) Found in 5.82 sec and 38.53MB; (ICV) Not found, spurious fault
	v15:	(IM) Found in 6.11 sec and 39.20MB
	v20:	(ICN) Found in 29.63 sec and 299.44MB

VI. CONCLUSIONS

We proposed a modular framework for verifying implementations and usage of reentrant locks in concurrent programs. Our approach decouples the verification of lock behavior from the verification of thread behaviors using lock interface machines. Our experiments demonstrate that our approach is capable of verifying realistic safety critical applications effectively.

REFERENCES

[1] E. Ábrahám-Mumm, F. S. de Boer, W. P. de Roever, and M. Steffen. Verification for java's reentrant multithreading concept. In *FOSSACS 2002*, pages 5–25.

[2] A. Betin-Can and T. Bultan. Verifiable concurrent programming using concurrency controllers. In *ASE 2004*, pages 248–257.

[3] A. Betin-Can and T. Bultan. Highly dependable concurrent programming using design for verification. *Formal Aspects of Computing*, 19(2):243–268, 2007.

[4] A. Betin-Can, T. Bultan, M. Lindvall, S. Topp, and B. Lux. Application of design for verification with concurrency controllers to air traffic control software. In *ASE 2005*, pages 14–23.

[5] A. Betin-Can, T. Bultan, Mikael Lindvall, Benjamin Lux, and Stefan Topp. Eliminating synchronization faults in air traffic control software via design for verification with concurrency controllers. *Automated Software Engineering*, 14(2):129–178, 2007.

[6] A. Chakrabarti, L. de Alfaro, T.A. Henzinger, M. Jurdziński, and F.Y.C. Mang. Interface compatibility checking for software modules. In *CAV 2002*, pages 428–441.

[7] R. DeLine and M. Fahndrich. Typestates for objects. In *ECOOP 2004*, pages 465–490.

[8] G. Delzanno. Automatic verification of parameterized cache coherence protocols. In *CAV 2000*.

[9] G. Dennis. TSAFE: Building a trusted computing base for air traffic control software, Master's Thesis, massachusetts institute of technology, 2003.

[10] H. Erzberger. Transforming the NAS: The next generation air traffic control system. In *International Congress of the Aeronautical Sciences (ICAS) 2004*.

[11] C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN 2003*, pages 213–224.

[12] C. Haack, M. Huisman, and C. Hurlin. Reasoning about java's reentrant locks. In *APLAS 2008*, pages 171–187.

[13] B. Jacobs, R. Leino, F. Piessens, and W. Schulte. Safe concurrency for aggregate objects with invariants. In *SEFM 2005*, pages 137–147.

[14] D. Lea. *Concurrent Programming in Java*. Addison-Wesley, Reading, Massachusetts, 1999.

[15] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI 2007*, pages 446–455.

[16] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS 2005*, pages 93–107.

[17] S. Qadeer and D. Wu. Kiss: keep it simple and sequential. In *PLDI 2004*, pages 14–24.

[18] B. Sarna-Starosta, R. E. K. Stirewalt, and L. K. Dillon. A model-based design-for-verification approach to checking for deadlock in multi-threaded applications. In *SEKE 2006*, pages 120–125, 2006.

[19] B. Sarna-Starosta, R. E. K. Stirewalt, and L. K. Dillon. A model-based design-for-verification approach to checking for deadlock in multi-threaded applications. *International Journal of Software Engineering and Knowledge Engineering*, 17(2):207–230, 2007.

[20] `java.util.concurrent` package. http://g.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html.

[21] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, 2003.

[22] T. Yavuz-Kahveci, C. Bartzis, and T. Bultan. Action language verifier, extended. In *CAV 2005*, pages 413–417.

[23] T. Yavuz-Kahveci and T. Bultan. Specification, verification, and synthesis of concurrency control components. In *ISSTA 2002*, pages 169–179, 2002.