Multiplexing of Partially Ordered Events

Colin Campbell[1], Margus Veanes[1], Jiale Huo[2⋆], and Alexandre Petrenko[3]

[1] Microsoft Research, Redmond, WA, USA
colin,margus@microsoft.com
[2] McGill University, Montreal, Quebec, Canada
jiale.huo@mail.mcgill.ca
[3] Centre de Recherche Informatique de Montreal, Quebec, Canada
petrenko@crim.ca

Abstract. This paper introduces a method to correctly order events in model-based testing for concurrent systems, in particular multi-threaded programs, whose events are only partially ordered. For a sequential, centralized tester, we need to merge (local) traces of each component into a (global) trace of a system in such a way that the ordering constraints are observed. To this end, we instrument a multi-threaded program under test so that the order of lock events is visible. This additional information helps a so-called multiplexer to reconstruct a fully serial trace consistent with the partial order. We describe programs and the multiplexer as labeled transition systems and give pseudo-code of the algorithm implementing the latter. The implementation of the algorithm presented is used in an industrial context.

1 Introduction

Model-based conformance testing checks whether an implementation is behaviorally consistent with its specification. Formally, this check is performed with respect to a correctness criterion called conformance relation. Such testing is carried out by a tester or a testing tool. An industrial software test engineer usually writes a test harness to provide an interface (API) between the tester and the implementation under test (IUT), so that the two entities can interact with each other. The interface is symmetric in the sense that it specifies the methods that the tester can use to influence the IUT and the methods that the IUT can use to pass information back to the tester.

The tester uses a model or specification as a reference of the IUT's behavior. The verdict of a particular test run depends on whether the observed behavior conforms to the specified behavior or not. For sequential systems, such as single-threaded programs, events can be observed in the order they occur. In concurrent systems, such as multi-threaded programs and distributed systems, events of individual agents (an agent being a thread or, in distributed systems, a process) can still be observed in the order they occur, but there are typically many possible ways in which events of different agents can be interleaved. In this paper, we consider the problem that a sequential, centralizaed tester is used to test concurrent systems. Due to its sequential nature, the tester requires a

⋆ Part of the work was carried out during the author's internship at Microsoft Research, Redmond.

linearized view of all the events of all agents. Other approaches consider using several distributed testers to test concurrent systems, see e.g. [5].

A problem with the single tester scenario is that, even if all the events of all agents are totally ordered according to a timeline with sufficiently fine precision, the order in which the events are observed by the tester may still differ from the actual one due to buffering and communication delays. If the agents in the IUT interact and this interaction is important to the conformance relation, observing the events out of order may result in false positives (a correct IUT failing a test) or false negatives (a wrong IUT passing a test), rendering the conformance checking unsound. (On the other hand, if the agents do not interact, they can be tested independently or concurrently by independent testers, and the need to observe the events in the corret order does not arise in the first place.) Inter-agent communication usually imposes a partial order of events defined by constraints on message communication, e.g. send-events happen before corresponding receive-events. In multi-threaded programs, a partial order of events is defined by the access of shared resources. For example, a lock has to be released by a thread before it can be acquired by another thread. In general, an event of one agent may depend on an event of another agent and therefore cannot occur before the latter. The sequence of events observed by the tester must not violate the dependency among the events.

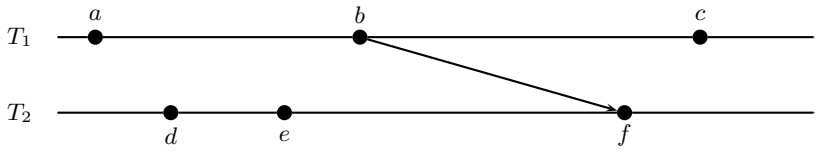

Fig. 1. Example of partially ordered events of a concurrent system.

Figure 1 illustrates events of a concurrent system with two agents using a space-time diagram [15]. The events of each agent are depicted as dots located on a per-agent timeline, on which an event x of the agent is drawn to the left of an event y of the agent if and only if x occurs before y, i.e. y depends locally on x. Inter-agent dependencies are indicated by arrows. In this system, event f of agent T_2 depends on event b of agent T_1, and therefore b must precede f. In this case the trace *daebfc* is consistent with the dependencies whereas the trace *daefbc* is not. In general, a trace is consistent with the partial order if and only if the trace represents an outcome of topological sorting, called a linearization, of the partial order.

In a system where all agents and all events are observed, it is straightforward to produce a linearization of the partial order of the events. For example, this is the case in a distributed system where each process is instrumented to produce unique send-events and receive-events of messages exchanged between the processes [11]. By using time stamps [14] all processes need not be observed but all communication relations must be augmented with a vector time stamps. When dealing with multi-threaded programs such instrumentations are often either impossible or undesirable. Threads do not directly communicate with each other, but synchronize through shared resourses, such as locks.

Lock events are not normally observable to the tester because they are internal to the implementation. The abstraction level at which they occur is lower than that of the model; as a consequence, lock events are not even mentioned in the model.

A naive attempt to reorder events in multi-threaded programs could be achieved by assigning a time stamp to each observable event with respect to a global clock and then sorting the events using the time stamps. However, modern computer hardware architectures may render the time stamping approach infeasible. For example, consider a program written for a multi-processor hardware architecture in which memory writes are local to each processor until an explicit memory-serialization operation occurs. Between two memory-serialization operations, the system never arrives in a single global state that can be seen uniformly by all processors. Hence, it seems impossible to use time stamps of a global clock to serialize the events occurring between two memory-serialization operations. Moreover, using a global clock may substantially alter the behavior being tested by introducing unwanted synchronization when the clock itself is a shared resource.

Another attempt to reconstruct a linearization from the observations would be to keep a centralized log of events [16]. In this scheme, each agent reports its events to a central, serialized log. Unfortunately, such a log introduces additional synchronization in multi-threaded programs because the very operation of writing into the log by each thread requires locking and unlocking the log. This additional synchronization could affect the possible behavior of the system and could eliminate certain errors. In other words, the instrumentation of the system would itself prevent some invalid behaviors from occurring. Undetected errors would occur once the system is no longer in "testing mode".

Our solution relies on additional assumptions about the implementation and instruments the implementation in such a way that the order in which locks are used becomes observable. We use a program called multiplexer that takes as its input sequences of events (with lock events included) of each agent and merges the event sequences into a single sequence that preserves the order of lock events. We show that if all the shared resources in the implementation are protected by locks then the merged event sequence is a valid linearization.

In 1978, Lamport described the inadequacy of using fully sequential time as a way to understand the runs of distributed systems [15]. His formulation of partially ordered distributed runs is consistent with the view presented in this paper, and like Lamport we use incrementing counters as a way to encode ordering constraints. However, the algorithm he presented focuses more on runtime synchronization (for example, as a way to solve the mutual exclusion problem), whereas our algorithm assumes proper synchronization in the concurrent system under test and validates its behavior with respect to a serial model of evolving system state.

The rest of the paper is organized as follows. Preliminaries are provided in Section 2. In Section 3, we formalize threads, shared resources and locks. Then, we describe the multiplexer formally in Section 4. The instrumentation of lock events is realized by extending the events with *usage counts* that indicate the order in which a lock is used by agents. We show that by using the multiplexer, the behavior of a multi-threaded program can be given a consistent serial interpretation. In Section 5, we outline the algorithm

underlying the multiplexer and mention its application in Section 6. Conclusions and discussions of future work are provided in Section 7.

2 Preliminaries

We use labeled transition systems (LTS) to describe the behavior of multi-threaded programs. A labeled transition system L has the following components: a nonempty set S of *states*; a nonempty subset S^{init} of S called *initial states*; a set Σ of *external actions*; a set Σ^{H} of *internal actions*, $\Sigma^{\text{H}} \cap \Sigma = \emptyset$; a *transition relation* $\delta \subseteq S \times (\Sigma \cup \Sigma^{\text{H}}) \times S$. L is denoted by the tuple $(S, S^{\text{init}}, \delta, \Sigma, \Sigma^{\text{H}})$. We sometimes index a component by L, unless L is clear from the context. Note that the sets of states and actions may be infinite. Given a transition $e = (s, a, t) \in \delta$; s is the *source* of e, t is the *target* of e, and a is the *label* of e; if $a \in \Sigma^{\text{H}}$ then e is an *internal* transition. The set of actions *enabled* or *defined* in a state s, denoted by $\mathit{En}(s)$, is the set of all labels of transitions whose source is s:

$$\mathit{En}(s) = \{a \in \Sigma \cup \Sigma^{\text{H}} | (\exists t \in S)(s, a, t) \in \delta\}.$$

A nonempty sequence α of external actions is called a *trace of L in state s_1* if there exist actions $a_1, \ldots, a_k \in \Sigma \cup \Sigma^{\text{H}}$ and states $s_1, \ldots, s_{k+1} \in S$ such that $(s_i, a_i, s_{i+1}) \in \delta$ for $1 \leq i \leq k$ and α is the projection of $a_1 \cdots a_k$ onto the set Σ. We write $\mathit{Tr}(s)$ to denote the set of all traces of L in state s; given $X \subseteq S$ we write $\mathit{Tr}(X)$ to denote $\bigcup_{s \in X} \mathit{Tr}(s)$, and we write $\mathit{Tr}(L)$ for $\mathit{Tr}(S^{\text{init}})$.

An LTS is *deterministic* if it has a single initial state, it has no internal transitions, and it has no transitions with the same source and label but distinct targets. If an LTS L is deterministic, it is convenient to view the transition relation as a partial function so that, given an action a that is enabled in a state s, $\delta_L(s, a)$ denotes the target of the transition in L whose source is s and label is a. For any LTS L there exists a deterministic LTS $\mathit{Det}(L)$ such that $\mathit{Tr}(L) = \mathit{Tr}(\mathit{Det}(L))$.

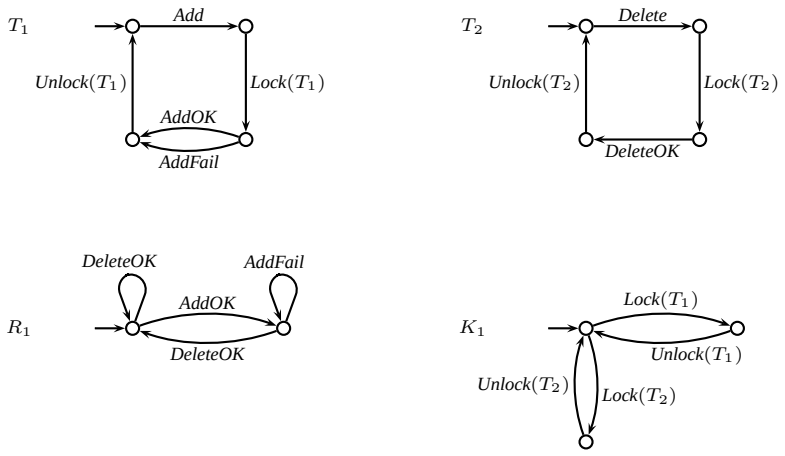

Fig. 2. Components of a system with two threads adding and deleting elements from a bag.

Example 1. The state machines in Figure 2 are deterministic LTSs. They model components of a multi-threaded program that adds and deletes elements from a shared bag R_1. For simplicity, the maximum capacity of the bag is restricted to a single element here but can easily be generalized to any number of elements. The bag is empty in the initial state and full in the other state. When empty, an element can be added to the bag, that is denoted by the action *AddOK*. Intuitively this action represents a successful attempt (method invocation) to add an element to the bag. The other action *AddFail* represents a failing attempt to add an element to the bag. Deleting an element from the bag always succeeds, even if there is nothing to delete. K_1 models a lock that protects the bag; it can be acquired (locked) and released (unlocked) by the two threads T_1 and T_2. T_1 models a thread executing a function *Add*. After *Add* is called, the thread acquires the lock K_1. It then either successfully adds an element or fails to add an element to the bag. This nondeterminism is resolved by the state of the bag (whether it is full or not). Finally, the lock is released and the behavior is repeated. T_2 models a thread that deletes elements from the bag.

Parallel composition of LTSs formalizes the interaction of several systems. In a composition of two LTSs the two systems will synchronize on shared external actions, and asynchronously interleave all other actions. Let $L_1 = (S_1, S_1^{\text{init}}, \delta_1, \Sigma_1, \Sigma_1^{\text{H}})$ and $L_2 = (S_2, S_2^{\text{init}}, \delta_2, \Sigma_2, \Sigma_2^{\text{H}})$ be two LTSs such that $\Sigma_i^{\text{H}} \cap \Sigma_j = \emptyset$, The (*parallel*) *composition* of L_1 and L_2 is an LTS $L_1 \parallel L_2 = (S, S^{\text{init}}, \delta, \Sigma, \Sigma^{\text{H}})$ where

- $S^{\text{init}} = S_1^{\text{init}} \times S_2^{\text{init}}$,
- $\Sigma = \Sigma_1 \cup \Sigma_2$, $\Sigma^{\text{H}} = \Sigma_1^{\text{H}} \cup \Sigma_2^{\text{H}}$,

and S is the smallest set of states and δ the smallest transition relation such that

- $S^{\text{init}} \subseteq S \subseteq S_1 \times S_2$,
- $a \in \Sigma_1 \cap \Sigma_2, \langle s_1, s_2\rangle \in S, (s_1, a, t_1) \in \delta_1, (s_2, a, t_2) \in \delta_2 \Rightarrow$
  $\langle t_1, t_2\rangle \in S, (\langle s_1, s_2\rangle, a, \langle t_1, t_2\rangle) \in \delta$,
- $a \in \Sigma_1^{\text{H}} \cup (\Sigma_1 - \Sigma_2), \langle s, u\rangle \in S, (s, a, t) \in \delta_1 \Rightarrow \langle t, u\rangle \in S, (\langle s, u\rangle, a, \langle t, u\rangle) \in \delta$,
- $a \in \Sigma_2^{\text{H}} \cup (\Sigma_2 - \Sigma_1), \langle u, s\rangle \in S, (s, a, t) \in \delta_2 \Rightarrow \langle u, t\rangle \in S, (\langle u, s\rangle, a, \langle u, t\rangle) \in \delta$.

Let $L = (S, S^{\text{init}}, \delta, \Sigma, \Sigma^{\text{H}})$ be an LTS. Let $B \subseteq \Sigma$. The LTS obtained by *internalizing* or *hiding* all the actions in B is the LTS $\mathit{Hide}[B](L) = (S, S^{\text{init}}, \delta, \Sigma - B, \Sigma^{\text{H}} \cup B)$. It is often convenient to assume, without loss of generality, that there is a single internal action τ, i.e., $\Sigma_L^{\text{H}} = \{\tau\}$, since the distinction of internal actions is unimportant in the definition of traces. We use $\mathit{DH}[B](L)$ as a shorthand for $\mathit{Det}(\mathit{Hide}[B](L))$.

Example 2. Consider the LTSs in Figure 2. $\mathit{DH}[\Sigma_{K_1}](T_1 \parallel T_2 \parallel R_1 \parallel K_1)$ is shown in Figure 3, where $\Sigma_{K_1} = \{\mathit{Lock}_{K_1}(T_1), \mathit{Unlock}_{K_1}(T_1), \mathit{Lock}_{K_1}(T_2), \mathit{Unlock}_{K_1}(T_2)\}$. Usually lock events are considered to be internal, so they are hidden in the composition.

Similar to [12], we use a *renaming* operator ‘$'$’ for the purpose of reusing the external actions of an LTS. The renaming operator is a bijection on actions. We lift the operator to sets of actions: for an action set A, $A' = \{a' | a \in A\}$. Given an LTS L we write L' for the LTS where are all actions in L have been renamed.

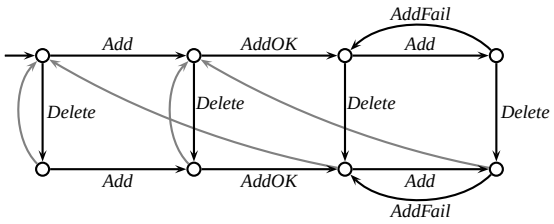

Fig. 3. The composition of the LTSs in Figure 2. Gray unlabeled arrows correspond to *DeleteOK*-transitions.

3 System modeling

We use LTSs to model multi-threaded programs. A thread is a sequential process modeled as an LTS. Two threads are *disjoint* if they do not share any actions. We consider a fixed collection *Threads* of n pairwise disjoint threads T_i for $1 \leq i \leq n$.

A *shared resource* is an LTS that models a state variable whose value is updated or read by threads. We consider a fixed collection *Resources* of m pairwise disjoint shared resources R_i, such that $\Sigma_{R_i} \subseteq \bigcup_{T \in \mathit{Threads}} \Sigma_T$, for $1 \leq i \leq m$.

Threads can communicate with each other through shared resources, but shared resources do not communicate with each other. For example, R_1 in Figure 2 is a shared resource.

A *lock* is a special type of shared resource that protects access to other shared resources. We model a lock K as a resource shared among the threads as follows.

$$
\begin{aligned}
S_K &= (\{\mathit{locked}_K\} \times \mathit{Threads}) \cup \{\mathit{unlocked}_K\},\\
S_K^{\text{init}} &= \{\mathit{unlocked}_K\},\\
\Sigma_K &= \{\mathit{Lock}_K(T) | T \in \mathit{Threads}\} \cup \{\mathit{Unlock}_K(T) | T \in \mathit{Threads}\},\\
\delta_K &= \{(\mathit{unlocked}_K, \mathit{Lock}_K(T), \langle \mathit{locked}_K, T\rangle) | T \in \mathit{Threads}\} \cup\\
&\quad \{(\langle \mathit{locked}_K, T\rangle, \mathit{Unlock}_K(T), \mathit{unlocked}_K) | T \in \mathit{Threads}\}.
\end{aligned}
$$

We consider a fixed collection *Locks* of l pairwise disjoint locks K_i for $1 \leq i \leq l$. For example, K_1 in Figure 2 is a lock.

This notion of locks does not allow a lock being acquired more than once without being released first. In some programming languages, such as C#, a thread can acquire a lock more than once, but it has to release the lock for the same number of times before the lock can be acquired by other threads. The locks as defined above are adequate for the purposes of this paper.

In the following, we use thread to refer to any program thread T_i above and we use shared resource only to refer to a shared resource that is not a lock.

Program threads, shared resources, and locks constitute a (*multi-threaded*) *program* $P = (\mathit{Threads}, \mathit{Resources}, \mathit{Locks})$. The *behavior* of P is described by the composition of the components denoted by $B(P)$. We hide *Lock* and *Unlock* actions in the composition, because they occur usually below the level of abstraction that is desired when viewing the composition, i.e. the lock events are not considered in the model.

$$B(P) \stackrel{\text{def}}{=} \mathit{DH}[\cup_{i=1}^{l} \Sigma_{K_i}](\|_{i=1}^{n}\ T_i\ \|_{i=1}^{m}\ R_i\ \|_{i=1}^{l}\ K_i).$$

Example 3. Consider the components in Figure 2 and let $P_1 = (\{T_1, T_2\}, \{R_1\}, \{K_1\})$. Figure 3 shows *Det*$(B(P_1))$. A practical concern when observing the behavior of such a system is to guarantee that the causal order of events is preserved. Since two threads are executing independently, it may happen for example that *AddFail* is observed after *DeleteOK*, resulting in an observed sequence *Add*, *AddOK*, *Add*, *Delete*, *DeleteOK*, *AddFail* that is not a trace of P_1, while in reality the trace *Add*, *AddOK*, *Add*, *Delete*, *AddFail*, *DeleteOK* happened.

The situation described in Example 3 can be formalized with the help of queues. Since threads are sequential processes, events from the same thread can be observed by a tester in the order they occur. Events from different threads could, however, have races. An event occurring earlier in one thread can be observed after an event occurring later in another thread. Recording of events can be formalized as buffering of events in thread-wise queues. Events are consumed in a random order from the queues by a tester. One can define queues similarly to those in [13], to model the effect of communication delay between the thread and the tester.

An *event queue* for a thread records events in the order produced by the thread and makes those events readable in FIFO order. Formally, given a thread $T \in$ *Threads*, Q_T is the following LTS:

$$\begin{aligned} S^{\text{init}}_{Q_T} &= \{\varepsilon\}, \\ S_{Q_T} &= (\Sigma_T)^*, \\ \Sigma_{Q_T} &= \Sigma_T \cup (\Sigma_T)', \\ \delta_{Q_T} &= \{(\alpha, a', \alpha a) | \alpha, \alpha a \in S_{Q_T}\} \cup \{(\alpha a, a, \alpha) | \alpha, \alpha a \in S_{Q_T}\}. \end{aligned}$$

Intuitively, a transition whose label is the renamed action a' corresponds to recording the event a in the queue, and a transition whose label is a corresponds to removing the recorded event a from the queue. Figure 4 illustrates an event queue of a thread with a single event a.

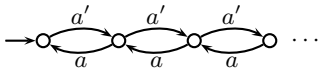

Fig. 4. An event queue for a thread with a single event a.

The queued behavior of a thread T can be described by composing T' with Q_T, hiding the shared actions, and making the result deterministic, i.e. the queued behavior is *DH*$[\Sigma'_T](T' \parallel Q_T)$. Unsurprisingly, *Tr*$(T) =$ *Tr*(*DH*$[\Sigma'_T](T' \parallel Q_T)$) because events from the same thread are observed in the order they occur.

Let $\mathbf{T}$, $\mathbf{R}$, $\mathbf{K}$ and $\mathbf{Q}$ denote the parallel compositions of threads, resources, locks, and queues respectively. For the program P as above, the external behavior of P composed with queues gives rise to the *queued behavior* $Q(P)$ of P,

$$Q(P) \stackrel{\text{def}}{=} \textit{DH}[\Sigma'_{\mathbf{T}}](B(P)' \parallel \mathbf{Q}) = \textit{DH}[\Sigma'_{\mathbf{T}}](\textit{DH}[\Sigma_{\mathbf{K}}](\mathbf{T} \parallel \mathbf{R} \parallel \mathbf{K})' \parallel \mathbf{Q}).$$

The set of traces of $Q(P)$ corresponds to the set of traces that may be observed by a tester. The set $Tr(Q(P))$ is a superset of $Tr(B(P))$, so a tester might observe some traces not in the original behavior of the program.

Example 4. In Figure 3, *Add AddOK Add Delete AddFail DeleteOK* is a trace of $B(P_1)$ for the program $P_1 = (\{T_1, T_2\}, \{R_1\}, \{K_1\})$ in Figure 2. This trace, however, could correspond to the following trace in $Q(P_1)$: *Add AddOK Add Delete DeleteOK AddFail* which is not in $B(P_1)$, as pointed out in Example 3.

4 Multiplexer

As described above, in order to avoid possible discrepancies between the observed and the actual behavior of a multi-threaded program, we use a multiplexer to create a linearization of the observed events. To this end, we instrument threads and locks to keep track of lock events with lock-wise counts, called *usage counts*. The usage count of a lock indicates the number of times the lock has been used. When the multiplexer reads events that have been logged in the queues, it keeps track of the usage counts and does not read a lock entry from a queue unless that entry has the expected usage count.

A lock K with a usage count is unlocked when the usage count is an even number; it is locked otherwise. Initially the usage count is 0 and K is unlocked. We model a lock K with a usage count as the following LTS:

$$
\begin{aligned}
S_K &= (\{\mathit{unlocked}_K\} \times \mathbb{N}^{\text{even}}) \cup (\{\mathit{locked}_K\} \times \mathit{Threads} \times \mathbb{N}^{\text{odd}}),\\
S_K^{\text{init}} &= \{\langle \mathit{unlocked}_K, 0\rangle\},\\
\Sigma_K &= \{\mathit{Lock}_K(T,i) | T \in \mathit{Threads}, i \in \mathbb{N}^{\text{even}}\} \cup\\
&\quad \{\mathit{Unlock}_K(T,i) | T \in \mathit{Threads}, i \in \mathbb{N}^{\text{odd}}\},\\
\delta_K &= \{(\langle \mathit{unlocked}_K, i\rangle, \mathit{Lock}_K(T,i), \langle \mathit{locked}_K, T, i+1\rangle) | T \in \mathit{Threads}, i \in \mathbb{N}^{\text{even}}\} \cup\\
&\quad \{(\langle \mathit{locked}_K, T, i\rangle, \mathit{Unlock}_K(T,i), \langle \mathit{unlocked}_K, i+1\rangle) | T \in \mathit{Threads}, i \in \mathbb{N}^{\text{odd}}\}.
\end{aligned}
$$

In order to observe the usage counts in traces, the usage counts are made an explicit part of the lock transition labels.

Example 5. Figure 5 shows the two threads T_1 and T_2 and the lock K_1 from Figure 2 extended with usage counts.

Given P, $\mathbf{T}$, $\mathbf{R}$, $\mathbf{K}$ and $\mathbf{Q}$ as above, the queued behavior of the program with lock events visible is described by the LTS $S(P)$,

$$S(P) \stackrel{\text{def}}{=} DH[\Sigma'_{\mathbf{T}}]((\mathbf{T} \parallel \mathbf{R} \parallel \mathbf{K})' \parallel \mathbf{Q})$$

The multiplexer communicates with $S(P)$ by reading events from the queues. Lock events are used to create a linearization of all the other events from different queues that respects the causal order of the events. If the first event in an event queue is a lock event, then the multiplexer checks whether its usage count is the expected one. If yes,

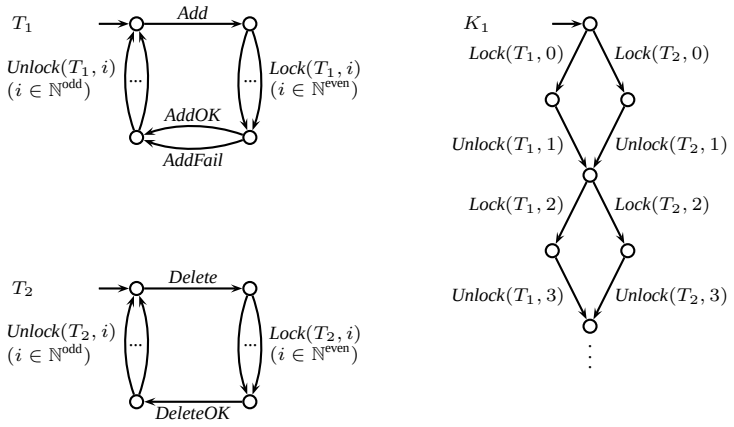

Fig. 5. Threads T_1 and T_2 and the lock K_1 from Figure 2 extended with usage counts.

then it deletes this event from the queue and increases the expected usage count of the lock; otherwise, it leaves the queue intact. If the first event in an event queue is not a lock event, which means that the event can be executed without violating the ordering constraint, the multiplexer can simply remove the event from the queue and puts it in the output queue read by the tester.

Formally, the multiplexer M is an LTS obtained from the composition $\mathbf{K}$ of locks with usage counts by adding self-loops for all non-locking actions:

$$\begin{aligned} S_M^{\text{init}} &= S_{\mathbf{K}}^{\text{init}}, \\ S_M &= S_{\mathbf{K}}, \\ \Sigma_M &= \Sigma_{\mathbf{T}}, \\ \delta_M &= \delta_{\mathbf{K}} \cup \{(s, a, s) | a \in \Sigma_M - \Sigma_{\mathbf{K}}, s \in S_M\}, \end{aligned}$$

The *multiplexed behavior* $M(P)$ of P is the composition of the queued behavior of P with the multiplexer where locking actions are hidden,

$$M(P) \stackrel{\text{def}}{=} \mathit{DH}[\Sigma_{\mathbf{K}}](S(P) \parallel M).$$

With the help of the multiplexer, we want to ensure that the multiplexed behavior $M(P)$ is the same as the behavior of P, i.e., $\mathit{Tr}(B(P)) = \mathit{Tr}(M(P))$. In general, this is only true if shared resources are properly protected by locks.

Fig. 6. Threads T_1 and T_2 from Figure 2 after removal of lock events.

Example 6. Assume for a moment that the threads T_1 and T_2 in Figure 2 do not use locks. In Figure 6, we show the threads without lock events as threads T_3 and T_4, respectively. The behavior $B(P_2)$ of the program $P_2 = (\{T_3, T_4\}, \{R_1\}, \{K_1\})$ with R_1 as in Figure 2 and K_1 as in Figure 5, happens to be the same as the LTS in Figure 3. It is easy to see that $\mathit{Tr}(B(P_2)) \neq \mathit{Tr}(M(P_2))$ because the shared resource R_1 is not protected by a lock.

One can see that if an event a of one thread, say T_1, must precede an event b of another thread, say T_2, in $B(P)$, then there must be lock events between a and b that effectively enforce this order. Since a lock has to be released by a thread before it can be acquired by another, if there are lock events between a and b in that order, there must be an *Unlock* event from T_1 before a *Lock* event from T_2. It is intuitively clear that we only need to protect events of shared resources (e.g. thread-local events need no protection).

Let P be as above. We say that P is *lock-protected*, if every shared resource R is associated with a lock K_R and for every trace $\alpha a \beta \in \mathit{Tr}(P)$ and thread T, where $a \in \Sigma_T \cap \Sigma_R$, there is a *Lock* event $\mathit{Lock}_{K_R}(T, k)$ in α and a corresponding *Unlock* event $\mathit{Unlock}_{K_R}(T, k+1)$ in β for some k. In other words, P is lock-protected if there is a lock for each shared resource that assures exclusive access to that resource one thread at a time.

The following theorem shows that multiplexing does not affect the traces of lock-protected programs.

Theorem 1. *Given P as above. If P is lock-protected then $\mathit{Tr}(B(P)) = \mathit{Tr}(M(P))$.*

Proof (outline). We show first that $\mathit{Tr}(S(P) \parallel M) \subseteq \mathit{Tr}(T \parallel R \parallel K)$. Consider a trace α of $S(P) \parallel M$. From the construction of $S(P)$ it follows that all events of a given thread appear in the correct order in α as renamed events. The events from queues are merged arbitrarily in $S(P)$ so causal ordering constraints between events from different threads is not preserved. However, composition with M and the assumption of P being lock-protected excludes illegal interleavings of the queues so that α is again a possible trace of $\mathit{Tr}(T \parallel R \parallel K)$.

To see that $\mathit{Tr}(T \parallel R \parallel K) \subseteq \mathit{Tr}(S(P) \parallel M)$ consider a trace $u = b_1 b_2 \cdots \in \mathit{Tr}(T \parallel R \parallel K)$. There is the particular trace $b'_1 b_1 b'_2 b_2 \cdots \in \mathit{Tr}((T \parallel R \parallel K)' \parallel Q)$ corresponding to the special case when an event is removed from a queue immediately after it has been added to the queue, and thus $u \in \mathit{Tr}(S(P))$. Moreover, since the lock event ordering is not violated in u, $u \in \mathit{Tr}(S(P) \parallel M)$.

From $\mathit{Tr}(S(P) \parallel M) = \mathit{Tr}(T \parallel R \parallel K)$ follows that

$$\begin{aligned} \mathit{Tr}(B(P)) &= \mathit{Tr}(\mathit{Hide}[\Sigma_K](T \parallel R \parallel K)) \\ &= \mathit{Tr}(\mathit{Hide}[\Sigma_K](S(P) \parallel M)) = \mathit{Tr}(M(P)). \end{aligned}$$ □

5 Multiplexing algorithm

In this section we describe the multiplexing algorithm that underlies a multiplexer. To make the description precise, we use the modeling language AsmL [2] as pseudo-code to describe the algorithm.

The multiplexer reads events from input queues. Each queue is associated with a particular thread. The multiplexer merges the events into a possible linearization and stores the merged sequence in a designated output queue.

```
type Queue
var inQueues as Set of Queue
var outQueue as Queue
```

The elements in the queues are lock events and other observable events, called update events. Each lock event is associated with a given lock and a usage count for that lock. (Each lock event is further classified as either acquiring or releasing of the lock, but this distinction is irrelevant for the purposes of this description.) The thread operating on the lock is implied by the input queue from which the multiplexer reads the lock event.

```
type Lock
structure Event
  case LockEvent
    lock as Lock
    count as Integer
  case UpdateEvent
```

We assume that one can perform the following operations on a queue: add a new event at the end of the queue by invoking `Enqueue`; remove the first event by invoking `Dequeue`; check if the queue is empty by invoking `IsEmpty`; and get the first event from the queue by invoking `Head`.

```
class Queue
  IsEmpty() as Boolean
  Enqueue(event as Event)
  Dequeue()
  Head() as Event
```

The multiplexer keeps a map from locks to expected usage counts. Initially, the map is empty, so the expected usage count of each lock is set to 0.

```
locks as Map of Lock to Integer = {->}
GetLockCount(lock as Lock) as Integer
  if lock notin locks then return 0
  else return locks(lock)
IncrementLockCount(lock as Lock)
  if lock notin locks then locks(lock) := 1
  else locks(lock) := locks(lock) + 1
```

The main part of the algorithm is described by the following while loop. A nonempty input queue of events is chosen randomly. If the first event is a lock event with a matching expected usage count then the event is removed from the queue and the expected usage count is incremented. If the event is an update event it is removed from the input queue and appended at the end of the output queue. From the point of view of external behavior, lock events are internal and are therefore not added to the output queue but are used solely for the purposes of ordering the update events.

```
while true
  choose queue in inQueues where not queue.IsEmpty()
    let e = queue.Head()
    if e is LockEvent then
      if e.count = GetLockCount(e.lock) then
        queue.Dequeue()
        IncrementLockCount(e)
      else
        skip
    else
      queue.Dequeue()
      outQueue.Enqueue(e)
```

This description of the algorithm is simplified. The actual implementation of the multiplexer is itself multi-threaded, where the input queues may be updated while the multiplexer is running. Moreover, the number of input queues may grow or shrink dynamically as the number of threads changes.

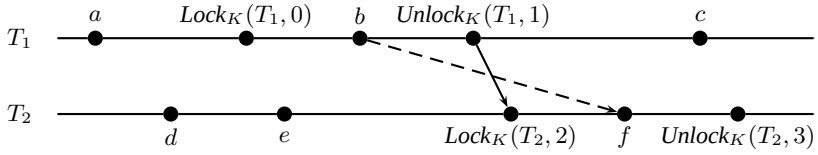

Fig. 7. Sample run of the threads in Figure 1.

Example 7. Figure 7 shows a possible run of the system in Figure 1. The event sequence of thread T_1 is $(a,\ Lock_K(T_1, 0),\ b,\ Unlock_K(T_1, 1),\ c)$, and the event sequence of thread T_2 is $(d,\ e,\ Lock_K(T_2, 2),\ f,\ Unlock_K(T_2, 3))$. The partial order of update events in the runs of the two threads depends on the total order of lock events associated with lock K. The solid arrow indicates that $Unlock_K(T_1, 1)$ happens before $Lock_K(T_2, 2)$. Consequently, event b must precede event f, as indicated with the dashed arrow. A possible event sequence produced by the multiplexer is $daebfc$. Notice that with the multiplexer, a tester always observes event b before event f since the order of update events is restrained by the order of lock events.

6 Application

The multiplexer is used together with the Spec Explorer tool for system-level conformance testing of multi-threaded and distributed systems. It is used by several Microsoft product groups that test highly concurrent subsystems of the forthcoming version of the Windows operating system. The Spec Explorer tool is briefly described in [8]. The tool is available from [1]. The threads or processes of the system under test produce thread-based event logs. These logs are serialized by the multiplexer into a single event trace.

The trace is fed into a conformance checking engine that checks whether the observed trace is valid with respect to a given specification or model. The model is described by a model program written in AsmL [9] or Spec# [3]. The use of a model program as a behavioral specification is explained in [4, 17]. The formal conformance relation that is checked between the model and the system under test is a variation of alternating refinement of interface automata [6, 7]. An event trace is viewed as a particular run of a game between two players: a tester (or testing tool) and a system under test. In this setting, the role of the multiplexer is to produce a serial view of the moves of the system, viewed as a single player, as a response to a move of the tester.

7 Conclusion

In this paper we considered model-based testing of multi-threaded programs with a single, sequential tester. Such a tester requires a linearized view of all the events that occurred in a given test run. We proposed a method for reordering of events from multiple threads so that partial order constraints concerning locks are not violated.

Our method requires some instrumentation of the program so that the partial order of lock events is used to help to reorder other events. We do not assume the existence of a globally visible clock, and our approach does not create additional synchronization between threads. In this sense, our method tries to avoid major impact on the system performance. We validated our approach, by modeling multi-threaded programs in terms of LTS, and formulated a sufficient condition in terms of lock usage.

Based on the assumption of lock-protection, our method ensures the soundness of a tester using the multiplexer. If a correct implementation is lock-protected, which is usually the case, the multiplexer can correctly reconstruct the events from the implementation, and the latter does not fail a test case derived from the model. On the other hand, if an implementation is correct but not lock-protected, possibly due to performance considerations, then the multiplexer can still produce some traces not belonging to the system. In this case, the correct implementation might fail a test.

The multiplexer is used together with the Spec Explorer tool for system-level conformance testing of multi-threaded and distributed systems. It is used by several Microsoft product groups that test highly concurrent subsystems of the forthcoming version of the Windows operating system.

As to the future work, we would like to extend our method to other applications where events have partial order constraints. For example, in communicating systems, a send event precedes the corresponding receive event and a request precedes the corresponding acknowledgment.

Also, the lock-protection condition looks a little too stringent. It could be relaxed by requiring lock-protection only when two events of a shared resources executed by different threads are totally ordered.

Moreover, the multiplexer could be extended to detect potential deficiencies of multithreaded programs, such as anti-patterns related to synchronization abuse and deadlock [10].

References

1. Spec Explorer. URL:http://research.microsoft.com/specexplorer, released January 2005.
2. AsmL. URL: http://research.microsoft.com/fse/AsmL/.
3. M. Barnett, R. Leino, and W. Schulte. The Spec# programming system: An overview. In M. Huisman, editor, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices: International Workshop, CASSIS 2004*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005.
4. A. Blass, Y. Gurevich, L. Nachmanson, and M. Veanes. Play to test. Technical Report MSR-TR-2005-04, Microsoft Research, January 2005. Extended version of a paper submitted to CAV'05.
5. L. Cacciari and O. Rafiq. Controllability and observability in distributed testing. *Inform. Software Technology*, 41:767–780, 1999.
6. L. de Alfaro. Game models for open systems. In N. Dershowitz, editor, *Verification: Theory and Practice: Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, volume 2772 of *LNCS*, pages 269 – 289. Springer, 2004.
7. L. de Alfaro and T. Henzinger. Interface automata. In *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of Software Engineering*, volume 26(5) of *ACM SIGSOFT Software Engineering Notes*, pages 109 – 120. ACM Press, 2001.
8. W. Grieskamp, N. Tillmann, and M. Veanes. Instrumenting scenarios in a model-driven development environment. *Information and Software Technology*, 46(15):1027–1036, December 2004.
9. Y. Gurevich, B. Rossman, and W. Schulte. Semantic essence of AsmL. In *Formal Methods for Components and Objects, Second International Symposium, FMCO 2003*, volume 3188 of *LNCS*, pages 240–259. Springer, 2004. Extended version to appear in special issue of *Theoretical Computer Science*, preliminary version available as Microsoft Research Technical Report MSR-TR-2004-27.
10. H. Hallal, E. Alikacem, P. Tunney, S. Boroday, and A. Petrenko. Antipattern-based detection of defficiencies in java multithreaded software. In *Proceedings of the Fourth International Conference on Quality Software (QSIC2004)*, Braunschweig, Germany, September 2004.
11. H. Hallal, S. Boroday, A. Ulrich, and A. Petrenko. An automata-based approach to property testing in event traces. In *Proceedings of the IFIP TC6/WG6.1 XV International Conference on Testing of Communicating Systems (TestCom 2003)*, volume 2644 of *LNCS*, pages 180–196. Springer, 2003.
12. J. Huo, R. Negulescu, and A. Petrenko. A study of robustness and delay-insensitivity of discrete action systems. Technical Report CRIM-03/04-02, Centre de Recherche Informatique de Montréal, Montreal, Quebec, Canada, 2003.
13. J. Huo and A. Petrenko. On testing partially specified IOTS through lossless queues. In *Proceedings of the 16th IFIP International Conference, TestCom 2004*, volume 2978 of *LNCS*, pages 76 – 94. Springer, 2004.
14. C. Jard. How to observe interoperability at the service level of protocols. In *7th IFIP WG6.1 International Workshop on Protocol Test Systems (IWPTS'94)*, Tokyo, Japan, November 1994.
15. L. Lamport. Time, clocks, and the orderings of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.
16. S. Tasiran and S. Qadeer. Runtime refinement checking of concurrent data structures. *Electronic Notes in Theoretical Computer Science*, 113:163–179, January 2005. Proceedings of the Fourth Workshop on Runtime Verification (RV 2004).
17. M. Veanes, C. Campbell, W. Schulte, and P. Kohli. On-the-fly testing of reactive systems. Technical Report MSR-TR-2005-05, Microsoft Research, January 2005.