Highly Dependable Concurrent Programming Using Design for Verification

Aysu Betin-Can[1] and Tevfik Bultan[2]

[1]Informatics Institute, Middle East Technical University, 06531, Ankara, Turkey
[2]Computer Science Department, University of California, Santa Barbara, CA 93106, USA

Abstract. There has been significant progress in automated verification techniques based on model checking. However, scalable software model checking remains a challenging problem. We believe that this problem can be addressed using a design for verification approach based on design patterns that facilitate scalable automated verification. In this paper, we present a design for verification approach for highly dependable concurrent programming using a design pattern for concurrency controllers. A concurrency controller class consists of a set of guarded commands defining a synchronization policy, and a stateful interface describing the correct usage of the synchronization policy. We present an assume-guarantee style modular verification strategy which separates the verification of the controller behavior from the verification of the conformance to its interface. This allows us to execute the interface and behavior verification tasks separately using specialized verification techniques. We present a case study demonstrating the effectiveness of the presented approach.

Keywords: model checking, interfaces, concurrent programming, synchronization, design patterns

1. Introduction

Automated software verification techniques based on model checking have improved significantly in recent years. When combined with the increasing computing power, these techniques are capable of analyzing complex software systems as demonstrated by numerous case studies. However, most applications of software model checking rely on a reverse engineering step in which either user guidance [24] or static analysis techniques [3, 25] (or both [11]) are used to rediscover some information about the software that may be known to its developers at design time. We believe that a *design for verification* approach that enables software developers to document the design decisions that can be useful during verification, can improve the scalability and therefore the applicability of model checking techniques significantly [8].

In this paper, we present a design for verification approach for synchronization in concurrent Java programs. Reliable concurrent programming is especially important for Java programmers since threads are an integral part of Java. Efficient 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. [29].

We present a behavioral design pattern called *concurrency controller pattern* for implementing synchronization

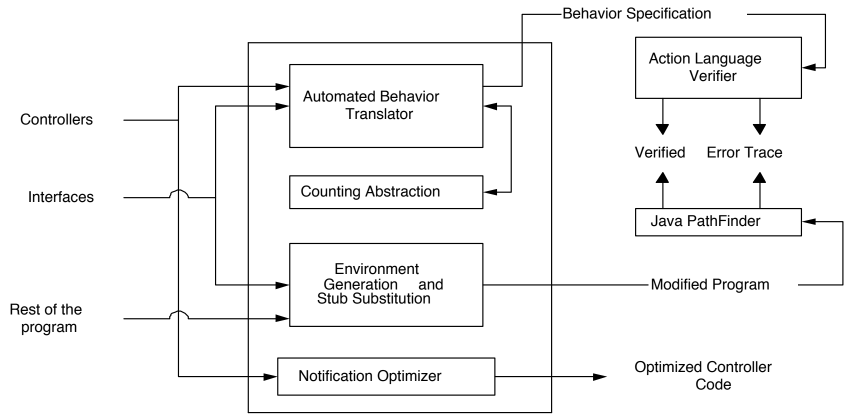


Fig. 1. System architecture overview

policies that coordinate interactions among multiple threads [5]. Concurrency controller pattern facilitates modular specification and verification, and also provides support for model extraction and environment generation steps required for model checking. In this pattern, a concurrency controller class encapsulates the variables and actions required for concurrency control and specifies the synchronization policy. We show that both safety and liveness properties of concurrency controllers can be automatically verified using a modular verification approach. We use stateful interfaces to decouple a controller's behavior from its environment. This decoupling enables an assume-guarantee style modular verification strategy which separates verification of the controller behavior (behavior verification) from the verification of the conformance of the threads to the controller interface (interface verification). The modularity and the specialization of the verification tasks are crucial for the scalability of our approach.

We show that concurrency controllers can be automatically optimized using the specific notification pattern. In the presented approach, programmers do not use the error-prone synchronization statements: `synchronized`, `wait`, `notify` and `notifyAll`. Synchronization statements are either provided in the pre-defined classes, or generated automatically during optimization.

An overview of our framework is illustrated in Fig. 1. A program written based on the concurrency controller pattern consists of controllers, their interfaces, and the rest of the program. These components are shown on the left side of Fig. 1. The tools we built to support our design for verification approach are shown in the main box. For behavior verification, a controller and its interface are given to our automated behavior translator tool. The behavior of each controller is verified separately. The translator outputs a behavioral specification of the controller in the input language of the infinite state model checker Action Language Verifier (ALV) [9, 41]. This specification is verified with respect to user defined properties. The result of this phase is either a certification that the controller satisfies the user defined properties, or a counter-example error trace demonstrating the violation of a property. In addition, we provide a counting abstraction module to verify the behavior of a concurrency controller for arbitrary number of threads. For the interface verification, the input is the controller interfaces and the rest of the program. The model checker we use for interface verification is Java PathFinder (JPF) [39]. Interface verification is performed on each concurrent thread separately. For this purpose, the interactions among the threads are abstracted by stub substitution and by replacing controllers with their interfaces. We also use JPF's nondeterminism utilities to create the environment of the threads. These modified programs are given to JPF to check the conformance of a thread to the controller interfaces. The result of the interface verification is either an assurance that the threads obey the controller interfaces, or an error trace showing the violation in the implementation with respect to the controller interfaces. In addition to the verification processes, our framework provides a notification optimizer that eliminates unnecessary context switches among concurrent threads using the specific notification pattern [10].

The rest of the paper is organized as follows. Section 2 introduces the concurrency controller pattern. Section 3 presents the formal models for the verification framework presented in this paper. Section 4 discusses the verification techniques for behavior and interface verification. The effectiveness of the proposed framework is investigated in a case study described in Section 5. Section 6 presents the related work, and Section 7 concludes the paper.

2. Concurrency Controller Pattern

Our goal in this work is to provide a design for verification approach for highly dependable synchronization in multi-threaded programs that have a set of concurrently accessed shared data and that require conditional waits and notifications for synchronization. In order to achieve this goal we developed a design pattern that resolves the following design forces:

- *The implementation should be verifiable*. There should be a scalable automated verification framework which ensures that the synchronization policies are correct with respect to desired safety and liveness properties. In recent years, there has been considerable progress in automated verification techniques for concurrent systems based on model checking. It should be possible to leverage this technology for the verification of synchronization policies.
- *The implementation should avoid common concurrent programming errors*. Usage of error-prone synchronization statements such as `synchronized`, `wait`, `notify`, and `notifyAll` should be avoided to prevent common programming errors such as nested monitor lockouts, missed or forgotten notifications, and slipped conditions [29].
- *The synchronization policy should be pluggable*. Encapsulating the synchronization policy with the implementation of the shared data classes downgrades the reusability of the code. It is time consuming and error-prone to modify the code to support new synchronization policies.
- *Shared data classes should be maintainable*. Encapsulating the implementation of the shared data with the synchronization operations makes modification of the shared data classes difficult. Managing synchronization among multiple threads and updating the states of the shared objects are separate concerns and it should be possible to modify them separately.
- *There should be an efficient mechanism to prevent unnecessary context-switch among threads*. The specific notification pattern [10] avoids context-switch overhead through multiple condition variables, multiple locks and notifications. However, the correct usage of these multiple locks and dependency analysis for correct notification is not easy to implement.

The concurrency controller pattern presented in this paper resolves these design forces. In the concurrency controller pattern, synchronization policies are implemented using guarded commands preventing error-prone usage of synchronization statements. The concurrency controller pattern separates the synchronization operations from the operations that change the shared object's internal state. This decoupling makes the synchronization policy pluggable and improves the maintainability of the code. We exploit this decoupling by using a modular verification technique based on controller interfaces. The modularity improves the efficiency of the verification process and enables us to verify large systems by utilizing different verification techniques such as infinite state symbolic model checking and explicit state model checking with their associated strengths. The concurrency controller pattern also resolves the difficulty in efficient implementation of synchronization policies since we provide an automated optimization technique based on the specific notification pattern.

Fig. 2 shows the class diagram for the concurrency controller pattern. The `ControllerInterface` is a Java interface that defines the names of the actions available to the threads. The `Controller` class specifies the synchronization policy. Multiple threads use an instance of the `Controller` class to coordinate their access to shared data. The `ControllerStateMachine` class is the controller interface that specifies the order of actions that can be executed by the threads. This class has an instance of the `StateMachine` class that supports specification of finite state machines and can be used as is. `SharedInterface` is the Java interface for the shared data. The actual implementation of the shared data is the `Shared` class. The `SharedStub` class specifies the constraints on accessing the shared data based on the interface states of the concurrency controller. We will explain these classes in more detail below.

2.1. Concurrency Controller Behavior

When applying the concurrency controller pattern, the `Controller` class in Fig. 2 is used for defining the behavior of the concurrency controller (i.e., this is the class where the synchronization policy is defined). The variables of the `Controller` class store only the state information required for concurrency control. Each action of the `Controller` class is associated with an instance of the `Action` class and consists of a set of guarded commands. The code for the `Action` class is the same for each controller implementation. To write a concurrency controller class based on the pattern in Fig. 2, a developer only needs to write the constructor for the `Controller` class, in which a set of guarded commands is defined for each action. Each method in the controller just calls the `blocking` or `nonblocking` method

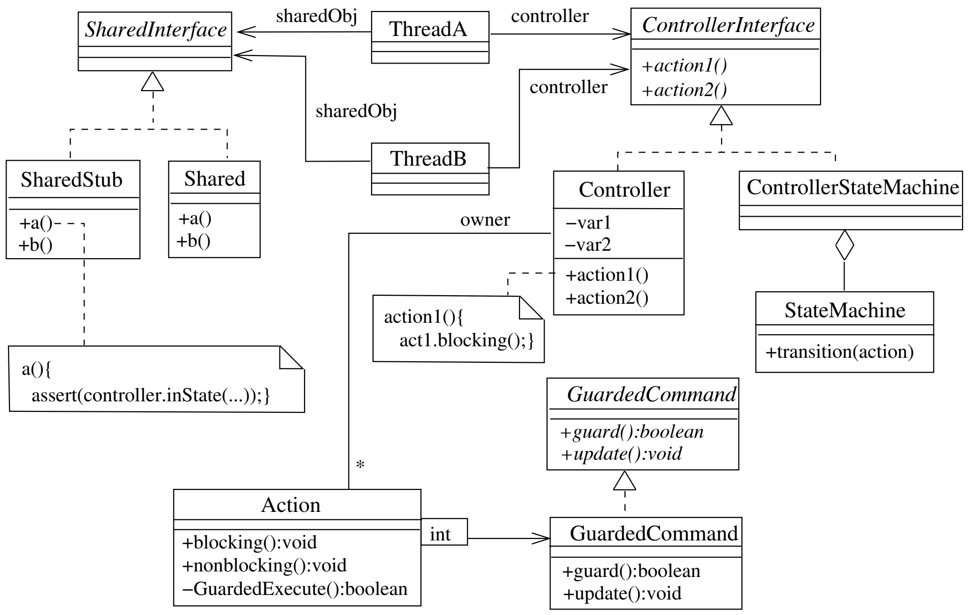


Fig. 2. Concurrency controller pattern class diagram

```
class BBMutexController implements BBMutex{
 boolean busy; int count; final int size;
 final Action act_consume_acquire, act_produce_acquire, act_release;
 BBMutexController(int size){
  count=0; busy=false; this.size=size;
  act_consume_acquire=new Action(this,new GuardedCommand(){
    public boolean guard(){ return count>0 && !busy;}
    public void update(){count=count-1; busy=true;}}}
  act_produce_acquire=new Action(this,new GuardedCommand(){
    public boolean guard(){ return count<size && !busy;}
    public void update(){count=count+1; busy=true;}}}
  act_release=new Action(this,new GuardedCommand(){
    public boolean guard(){ return busy;}
    public void update(){busy=false;}}}
 }
 public void consume_acquire(){act_consume_acquire.blocking();}
 public void produce_acquire(){act_produce_acquire.blocking();}
 public void release(){act_release.blocking();}
}
```

Fig. 3. BB-MUTEX implementation based on the concurrency controller pattern

of the corresponding action. A blocking action causes the calling thread to wait until one of the guarding conditions becomes true whereas a nonblocking action does not cause the calling thread to wait.

As an example, consider a class called `Buffer` which is a data buffer implementation with two methods `void put(Object obj)` and `Object take()` where the `take` operation removes an item from the buffer and the `put` operation puts an item to the buffer. Assume that this data buffer is going to be shared among multiple threads. Hence, we need to implement a synchronization policy to coordinate the accesses to this shared buffer. One possible synchronization policy is as follows: The accesses to the shared buffer should be mutually exclusive. In addition, the synchronization policy should ensure that a thread that wants to put an item to the buffer will wait while the buffer is full. Similarly, a thread that wants to take an item from the buffer will wait while the buffer is empty. We will call this synchronization policy BB-MUTEX, a bounded buffer synchronized with a mutex lock.

The BB-MUTEX synchronization policy can be implemented as a concurrency controller using three variables and three actions. The variables are `busy`, `count`, and `size`. Here, `busy` denotes whether there is a thread in the critical section, `count` denotes the number of items in the buffer, and `size` denotes the size of the buffer. The controller actions are `consume_acquire`, `produce_acquire`, and `release`. The `BBMutexController` class in Fig. 3 shows an implementation of the BB-MUTEX controller.

```
public class Action{
 protected final Object owner;
 private final Vector gcV;
 public Action(Object owner, Vector gcs){...}
 public Action(Object owner, GuardedCommand gc){...}
 private boolean GuardedExecute(){
  boolean result=false;
   for(int i=0; i<gcV.size(); i++)
    try{
     if(((GuardedCommand)gcV.get(i)).guard()){
      ((GuardedCommand)gcV.get(i)).update();
      result=true; break; }
    }catch(Exception e){}
  return result;
 }
 public boolean nonblocking(){
  synchronized(owner) {
   boolean result=GuardedExecute();
   if (result) owner.notifyAll();
   return result; }
 }
 public void blocking(){
  synchronized(owner) {
   while(!GuardedExecute()) {
    try{owner.wait();}
    catch (Exception e){} }
   owner.notifyAll(); }
 }
}
```

Fig. 4. Action class

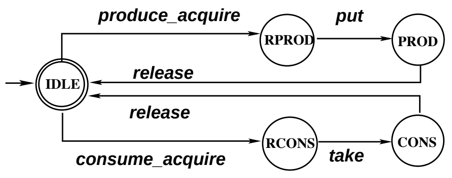


Fig. 5. Concurrency controller interface for BB-MUTEX

We specify the behavior of a concurrency controller in a guarded command style similar to that of CSP [26]. Since the Java language does not have a guarded command structure, we provide the `GuardedCommand` interface and the `Action` class which can be used as is without any modifications. Each instance of the `Action` class has a vector of guarded commands that defines its behavior. The code for the `Action` class is given in Fig. 4.

The `Action` class implements the guarded command semantics as follows. The `Action` class has three significant methods. The `GuardedExecute` method is used for executing one of the guarded commands of the action. If all the guards evaluate to false, then this method returns `false`. The execution of a blocking action is implemented by the `blocking` method. When a thread calls a blocking action, it has to execute a guarded command. Therefore, if the `GuardedExecute` method does not execute one of the guarded commands, then the thread waits in a loop, until it is notified by another thread. The execution of a nonblocking action is implemented by the `nonblocking` method. This method calls the `GuardedExecute` and notifies the other threads if a guarded command is executed. Note that, a nonblocking action does not cause the calling thread to wait. A nonblocking action returns `true` if a guarded command is successfully executed and returns `false` if the guards of all its guarded commands are false.

2.2. Concurrency Controller Interface

The interface of a concurrency controller defines the acceptable call sequences for each thread that uses the controller. Note that, controller interfaces have states and cannot be specified as Java interfaces. In the concurrency controller pattern, we specify the controller interfaces as Java classes. The `ControllerStateMachine` class in Fig. 2 defines the controller interface. This class encodes the state machine defining the allowed action call sequences. To encode the state machine we provide a `StateMachine` class, which is a finite state machine implementation and can be used as is. The `ControllerStateMachine` has the same set of methods as the concurrency controller itself. When a method

```
public BBMutexStateMachine implements BBMutex{
 StateMachine sm;
 public final static int IDLE=0, RPROD=1, PROD=2;
 public final static int RCONS=3, CONS=4;
 public BBMutexStateMachine(int sz){
   sm=new StateMachine(5);
   sm.initial(IDLE);
   sm.addTransition("produce_acquire",IDLE,RPROD);
   ...
 }
 public void produce_acquire(){
  sm.transition("produce_acquire");}
 ...
}
```

(a)

```
public class BufferStub implements Buffer{
 ....
 public Object take(){
  assert(controller.inState(BBMutexStateMachine.RCONS));
  controller.change(BBMutexStateMachine.CONS);
 }
 public Object put(){
  assert(controller.inState(BBMutexStateMachine.RPROD));
  controller.change(BBMutexStateMachine.PROD);
 }
 ...
}
```

(b)

Fig. 6. Parts of the controller interface implementation for BB-MUTEX

of the controller interface is called, the transition method of the `StateMachine` with the corresponding action name is invoked. This `transition(action)` method first executes an assertion which checks that the current state is a state where the `action` can be executed, and then sets the current state to the target state of that transition.

The controller interface is also used to specify when the methods of the shared data objects can be executed. Recall the bounded buffer example discussed above. One requirement, for example, should be that the synchronization method `consume_acquire` should be called before the `take` method of the buffer is called. In the concurrency controller pattern, these constraints are specified as assertions in a data stub class, shown as `SharedStub` in Fig. 2.

The interface machine for the BB-MUTEX concurrency controller is shown in Fig. 5. This interface machine has five states: `IDLE`, `RPROD` (denoting that the corresponding thread is ready to produce an item), `PROD` (denoting that an item is produced), `RCONS` (denoting that the corresponding thread is ready to consume an item), and `CONS` (denoting that an item is consumed), with `IDLE` being the initial state. The interface state machine shows how the interface state changes when an action is executed. The BB-MUTEX controller interface, for example, states that a thread using the BB-MUTEX controller should only execute (i.e. call) the `release` action after executing the `produce_acquire` or the `consume_acquire` actions of the controller. It also states that the `take` method of the `Buffer` can only be executed immediately after executing the `consume_acquire` action and the `put` method of the `Buffer` can only be executed immediately after executing the `produce_acquire` action.

This controller interface can be defined with a `BBMutexStateMachine` class shown in Fig. 6(a) by using a `StateMachine` instance, and a `BufferStub` class to define when the methods of the `Buffer` should be invoked as shown in Fig. 6(b). In Fig. 6(a), the states `IDLE`, `RPROD`, `PROD`, `RCONS`, and `CONS` are encoded as integer constants. The transitions are defined in the constructor. A method of the `BBMutexStateMachine` class contains only the invocation of the corresponding transition in the state machine instance, e.g., the `produce_acquire` method invokes the transition from `IDLE` to `RPROD`. In Fig. 6(b), the methods of the `BufferStub` contains the assertions specifying the access rules of the method, e.g., the `take` method of the `Buffer` can be executed only in the `RCONS` state.

2.3. Optimizing Concurrency Controllers

Concurrency controllers written using the design pattern given in Fig. 2 may be inefficient because of the following reasons: 1) The pattern in Fig. 2 does not use specific notification, hence, after every state change in the concurrency controller all the waiting threads are awakened, increasing the context-switch overhead; 2) The inner classes used in the pattern in Fig. 2 and the large number of method invocations may degrade the performance. To solve both of these problems, we automatically optimize the concurrency controllers using a source-to-source transformation. The optimized controller class 1) uses the specific notification pattern [10], 2) does not have any inner classes, and 3) minimizes the number of method invocations. During this transformation, a notification dependency analysis is required to implement the specific notification pattern. Such an analysis can be difficult and complicated to do manually. Our optimizer uses the algorithm presented by Yavuz-Kahveci et al. [42] to compute these dependencies automatically. Fig. 7 is an excerpt from the optimized version of BB-MUTEX controller generated from the source given in Fig. 3.

```
class BBMutexController implements BBMutex{
 private boolean busy; private int count; private final int size;
 private Object Condconsume_acquire, Condproduce_acquire, Condrelease;
 public BBMutexController(int size){....}
 private boolean Guarded_consume_acquire(){
  boolean result=false;
  synchronized(this) {
   if(count>0 && !busy){
     count=count-1; busy=true; result=true;}
   else; }
  return result;
 }
 public void consume_acquire(){
  synchronized(Condconsume_acquire){
   while (!Guarded_consume_acquire()){
    try{ Condconsume_acquire.wait();
    } catch(InterruptedException e){}}}
  synchronized(Condrelease){Condrelease.notifyAll();}
 }
 ....
 private boolean Guarded_release(){
  boolean result=false;
  synchronized(this){
   if(busy){
      busy = false;result=true;}
   else;}
  return result;
 }
 public void release(){
  synchronized(Condrelease){
   while (!Guarded_release()){
    try{Condrelease.wait();
    }catch(InterruptedException e){}}}
  synchronized(Condproduce_acquire){Condproduce_acquire.notifyAll();}
  synchronized(Condconsume_acquire){Condconsume_acquire.notifyAll();}
 }
}
```

Fig. 7. BBMutexController class produced by optimization

3. Formal Models

In this section we present the formal models for our modular verification approach. We first define an interface model. Then, we introduce the formal semantics of the concurrency controllers. We continue with presenting an abstract model for concurrent and distributed programs. After the presentation of these models, we discuss the formal basis of our modular interface and behavior verification techniques.

3.1. Interface Model

The interface of a concurrency controller specifies the allowed sequencing of controller actions and shared data operations. An interface $I = (Q, q_0, \Sigma, \delta, F)$ is a finite state machine where Q is the finite set of states, $q_0 \in Q$ is the initial state, Σ is the input alphabet, $\delta : \Omega \to Q$ is the transition function, where $\Omega \subseteq Q \times \Sigma$ denotes the set of actions allowed in each state, and $F = \{q_0\}$ is the set of final states. An action sequence $w \in \Sigma^*$ is a legal sequence of the interface $I = (Q, q_0, \Sigma, \delta, F)$ if, starting from q_0 and executing I with respect to w, reaches a state in F.

Consider our running example where the BB-MUTEX controller is used for coordinating the concurrent accesses to the shared `Buffer`. The interface specification $I = (Q, q_0, \Sigma, \delta, F)$ for this controller is shown in Fig. 5. The set of interface states is $Q = \{$`IDLE, RPROD, RCONS, PROD, CONS`$\}$, the initial state is q_0 =`IDLE`, and the set of final states is $F = \{$`IDLE`$\}$. The input alphabet is $\Sigma = \{$`put, take, produce_acquire, consume_acquire, release`$\}$. The transition function δ for this interface is shown as arrows in Fig. 5.

Note that, since δ is a function, given a state q and an action σ there is only one next state. This determinism does not affect the expressiveness of the interface model since any nondeterministic finite state machine can be converted to an equivalent deterministic finite state machine.

3.2. Concurrency Controller Semantics

A concurrency controller is a tuple $CC = (\Gamma, IC, A, I)$, where Γ is the set of controller variables, IC is the initial condition, A is the set of actions, and I is the controller interface. For example, for the BB-MUTEX concurrency controller, $\Gamma = \{$ `busy`, `count`, `size` $\}$ and $A = \{$ `consume_acquire`, `produce_acquire`, `release` $\}$. The concurrency controller variables are private and can only be modified or accessed via the actions of CC. The initial condition denotes the initial values assigned to the concurrency controller variables in the constructor of the controller class. Formally, IC is a predicate on the concurrency controller variables in Γ, i.e., $IC : \prod_{\gamma\in\Gamma} \text{DOM}(\gamma) \rightarrow \{\text{TRUE}, \text{FALSE}\}$, where $\text{DOM}(\gamma)$ denotes the domain of the concurrency controller variable γ and $\prod_{\gamma\in\Gamma} \text{DOM}(\gamma)$ denotes the Cartesian product of the variable domains. For the BB-MUTEX $IC \equiv \neg\texttt{busy} \wedge \texttt{count}= 0$.

The semantics of a concurrency controller specification CC is a transition system $T(CC)(n) = (IT, ST, RT)$ where n is the parameter denoting the number of threads, ST is the set of states, $IT \subseteq ST$ is the set of initial states, and $RT \subseteq ST \times ST$ is the transition relation. The transition system $T(CC)(n)$ represents all possible behaviors of a controller object when it is shared among n threads and assuming that each thread uses the controller object according to its interface. The initial states correspond to the states of the controller at the end of the execution of the constructor method. The transition relation RT represents the behavior of the controller object by recording its state at the end of the execution of each controller method that corresponds to an action.

The set of states is defined as the Cartesian product of the concurrency controller variable domains and the states of the threads, $ST = \prod_{\gamma\in\Gamma} \text{DOM}(\gamma) \times \prod^n Q$. Note that, the state of a thread is represented by an interface state and there is one interface state per thread.

We introduce the following notation. Given a state $s \in ST$ and a controller variable $\gamma \in \Gamma$, $s(\gamma) \in \text{DOM}(\gamma)$ denotes the value of the variable γ in state s. Given a state $s \in ST$ and a set of variables $\Gamma' \subseteq \Gamma$, $s(\Gamma') \in \prod_{\gamma\in\Gamma'}$ denotes the projection of state s to the domains of the variables in Γ'. Finally, given a state s and a thread t, where $1 \leq t \leq n$, $s(Q)(t) \in Q$ denotes the state of thread t in s, and $s(Q - t) \in \prod^{n-1} Q$ denotes the projection of state s to the states of all the threads except thread t.

The initial states of the transition system $T(CC)(n)$ is defined as $IT = \{s \mid s \in ST \wedge IC(s) \wedge \forall 1 \leq t \leq n,\ s(Q)(t) = q_0\}$

The set of actions, A, specifies the behavior of the concurrency controller. Each action $a \in A$ consists of a set of guarded commands $a.GC$. Each action has a blocking/nonblocking tag. For each guarded command $gc \in a.GC$, guard $gc.g$ is a predicate on the variables Γ, $gc.g : \prod_{\gamma\in\Gamma} \text{DOM}(\gamma) \rightarrow \{\text{TRUE}, \text{FALSE}\}$. For each guarded command $gc \in a.GC$, the update $gc.u$ is defined on controller variables $gc.u = \prod_{\gamma\in\Gamma} \text{DOM}(\gamma) \rightarrow \prod_{\gamma\in\Gamma} \text{DOM}(\gamma)$.

Let $q \in Q$ be an interface state and $a \in A$ be an action. We will define a transition relation $RT_{(q,a)} \subseteq ST \times ST$, which corresponds to executing the action a at interface state q. We define $RT^u_{(q,a)}$, the transition relation when one guarded command is executed, as

$$\begin{aligned} RT^u_{(q,a)} = \ & \{(s, s') \mid s, s' \in ST \ \wedge\ (\exists gc \in a.GC, gc.g(s(\Gamma)) \ \wedge\ s'(\Gamma) = gc.u(s(\Gamma))) \\ & \wedge (\exists 1 \leq t \leq n, s(Q)(t) = q \ \wedge\ s'(Q)(t) = \delta(q, a) \ \wedge\ s'(Q - t) = s(Q - t))\} \end{aligned}$$

If the action a is blocking, then the transition relation for the tuple (q, a) is defined as $RT_{(q,a)} = RT^u_{(q,a)}$. If the action a is nonblocking, then $RT_{(q,a)} = RT^u_{(q,a)} \cup RT^{nb}_{(q,a)}$ where $RT^{nb}_{(q,a)}$ denotes the case where none of the guards of a evaluate to TRUE

$$\begin{aligned} RT^{nb}_{(q,a)} = \ & \{(s, s') \mid s, s' \in ST \ \wedge\ s'(\Gamma) = s(\Gamma) \ \wedge\ (\forall gc \in a.GC,\ \neg gc.g(s(\Gamma))) \\ & \wedge (\exists 1 \leq t \leq n, s(Q)(t) = q \ \wedge\ s'(Q)(t) = \delta(q, a) \ \wedge\ s'(Q - t) = s(Q - t))\} \end{aligned}$$

The transition relation RT of the transition system $T(CC)(n)$ is defined as $RT = \bigcup_{(q,a)\in\Omega} RT_{(q,a)}$ where Ω denotes the set of actions allowed in each interface state (see the definition of the interface transition function in Section 3.1).

We define execution paths of the transition system $T(CC)(n)$ based on RT as follows: An execution path $s_0, s_1, \ldots$ is a path such that $s_0 \in IT$ and $\forall i \geq 0,\ (s_i, s_{i+1}) \in RT$.

We use the temporal logic CTL [15] to state properties about concurrency controllers. Let AP denote the set of atomic properties, where a property $p \in AP$ is a predicate on the concurrency controller variables in Γ, $p : \prod_{\gamma\in\Gamma} \text{DOM}(\gamma) \rightarrow \{\text{TRUE}, \text{FALSE}\}$. A concurrency controller CC satisfies a CTL formula f, if and only if, $\forall n \geq 0$, all the initial states of the transition system $T(CC)(n)$ satisfy the formula f.

3.3. Program Model

In this section, we first introduce an abstract execution model for concurrent programs implemented using the concurrency controller pattern. Then, we define two projection functions on this model in order to explain the formal basis for the interface and behavior verification steps.

3.3.1. A Model for Concurrent Programs

We assume that, in a concurrent program P, there are three types of threads: 1) the main thread, 2) the threads that are created by other threads explicitly, and 3) the threads that are created implicitly by, for example, the Java Runtime Environment. In Java, an explicit thread is created with the invocation of the `start()` method of a class that extends `java.lang.Thread` or implements `java.lang.Runnable`. The implicit threads are the event thread (`java.awt.EventDispatchThread`) that dispatches the graphical user interface (GUI) events and the RMI threads that serve the remote method invocations. We assume that the runtime environment creates one implicit thread for each RMI connection.

In order to develop a formal model for concurrent programs we use the following assumptions: 1) The shared variables are known and are implemented based on the concurrency controller pattern. Hence, the only shared variables in the program are the concurrency controllers and the shared data protected by these controllers. A variable is shared if more than one thread accesses that variable during a program execution. Using an escape analysis [14, 7, 27] one can find the set of shared variables to confirm this assumption. 2) The shared data is not primitive type. Otherwise, the concurrency controller pattern cannot be applied since the pattern requires an interface machine for the shared data. Also, the fields of the shared data are private and the only way to access or modify these fields is through method calls. 3) Suppose a thread t uses more than one controller. We assume that t executes an action of a controller if it is at the initial interface state (which is the only final state) with respect to all other controllers. However, we allow composition of interfaces of concurrency controllers which then relaxes this restriction [4]. 4) To simplify the program model we assume that the methods of the shared data are basic methods, that is, they do not invoke other methods. This assumption can be relaxed too.

Before formulating a program configuration and execution, we define the program stores in a concurrent program. A shared store mapping is defined as $\rho \in \mathit{Sh} : \mathcal{V} \rightarrow \bigcup_{v \in \mathcal{V}} \text{DOM}(v) \cup \perp$ where $\mathcal{V}$ is the set of program variables that are accessed by more than one thread. Given $v \in \mathcal{V}$, a shared store mapping $\rho(v)$ returns the value of v if v is visible to more than one thread, and otherwise returns $\perp$. We assume that the mappings are correct with respect to types. I.e., for all $\rho \in \mathit{Sh}$ and $v \in \mathcal{V}$, $\rho(v) \in \text{DOM}(v) \cup \perp$. Based on our first assumption, the elements of $\mathcal{V}$ to be used in shared store mappings are known. Intuitively, ρ represents the shared stores in program P through which the concurrent threads communicate with each other. We define a local store mapping for a thread as follows. Given a thread t, the local store mapping of t is $\ell^t \in \mathit{Lcl}(t) : \mathcal{V} \rightarrow \bigcup_{v \in \mathcal{V}} \text{DOM}(v) \cup \perp$ where $\mathcal{V}$ is the set of program variables that are accessed by t. Given $v \in \mathcal{V}$, a local store mapping $\ell^t(v)$ returns the value of v if v is visible only to t and returns $\perp$ otherwise. We assume the local mappings are also correct with respect to types. Intuitively, ℓ^t represents the local stores of t that are accessed only by t. Suppose $\ell^t(v) \neq \perp$ and $\rho(v) = \perp$ at a program configuration, and v becomes visible to more than one thread at the next program configuration. Then, the shared store mapping becomes $\rho'(v) \neq \perp$ and the local store mapping becomes $\ell'^t(v) = \perp$.

Program Configurations: A configuration of a program implemented based on the concurrency controller pattern consists of a shared store mapping ($\rho \in \mathit{Sh}$), a local store mapping for each thread t ($\ell^t \in \mathit{Lcl}(t)$), and a control state for each thread t ($\alpha^t \in \mathit{Ctl}(t)$) which is the program counter of t. Hence, the set of program configurations is $\mathit{Sh} \times \prod_{t \in T}(\mathit{Lcl}(t) \times \mathit{Ctl}(t))$ where T is the finite set of threads. The number of threads in a program changes during the program execution with explicit thread creations and thread terminations. We represent the set of threads at a configuration as $c(T)$. We assume that all of the implicit threads are created at the beginning of the program execution. Then, the threads at the beginning of the program execution are: the main thread t_m, the event thread t_e and one thread for each RMI connection, $t_{r_1}, t_{r_2}, \ldots, t_{r_k}$. A program starts with an initial configuration $c_0 = (\rho_0, \ell_0^{t_m}, \alpha_0^{t_m}, \ell_0^{t_e}, \alpha_0^{t_e}$, $\ell_0^{t_{r0}}$, $\alpha_0^{t_{r1}}$, $\ell_0^{t_{r1}}$, $\alpha_0^{t_{r2}}$, $\ldots$, $\ell_0^{t_{r_k}}$, $\alpha_0^{t_{r_k}})$ where $\rho_0 \in \mathit{Sh}$ is the initial shared store mapping that returns $\perp$ for every variable, $\ell_0^{t_m} \in \mathit{Lcl}(t_m)$ is the initial local store mapping of the main thread t_m, $\alpha_0^{t_m} \in \mathit{Ctl}(t_m)$ is the initial control state of t_m, $\ell_0^{t_e} \in \mathit{Lcl}(t_e)$ is the initial local store mapping of the event thread t_e, $\alpha_0^{t_e} \in \mathit{Ctl}(t_e)$ is the initial control state of t_e, and, for $1 \leq j \leq k$, $\ell_0^{t_{r_j}} \in \mathit{Lcl}(t_{r_j})$ is the initial local store mapping of the RMI thread t_{r_j}, and $\alpha_0^{t_{r_j}} \in \mathit{Ctl}(t_{r_j})$ is the initial control state of t_{r_j}. The program configuration changes with an input event or with the execution of an operation by one of the concurrent threads.

Operations and Input Events: Given a thread t, we denote an operation that t can execute as op^t. There are five types of operations: local operations, shared operations, thread creation, thread termination, and environment interaction operations. We will use *op* to denote any kind of operation.

Local operations are the operations that change only a thread's local store mapping and its control state, and that are not environment interaction operations. Given a thread t, a local operation performed by t is defined as $lop^t : \mathit{Lcl}(t) \times \mathit{Ctl}(t) \rightarrow \mathit{Lcl}(t) \times \mathit{Ctl}(t)$. We do not define the granularity of local operations in our formal model. As long as it does not access or modify any shared variable, a local operation may correspond to a single statement, a sequence of statements or a whole method execution.

Shared operations are the operations through which threads interact with each other. These operations are either a read from a shared variable or a write to a shared variable. Recall that a shared variable can be either a concurrency controller instance or a shared data protected by a controller. Based on the concurrency controller pattern, all of the shared operations are known beforehand. A shared operation performed by t is defined as $sop^t : \mathit{Sh} \times \mathit{Ctl}(t) \rightarrow \mathit{Sh} \times \mathit{Ctl}(t)$.

The thread creation operation is a special type of interaction used for creating explicit threads. Let t be the thread that creates the explicit thread t' with the operation $tcop^t$. We define this type of operation as $tcop^t : \mathit{Sh} \times \mathit{Lcl}(t) \times \mathit{Ctl}(t) \rightarrow \mathit{Sh} \times \mathit{Lcl}(t) \times \mathit{Ctl}(t) \times \mathit{Lcl}(t') \times \mathit{Ctl}(t')$. With a thread creation operation there are zero or more program variables that become visible to more than one thread. Let $tcop^t(\rho_i, \ell^t_i, \alpha^t_i) = (\rho_{i+1}, \ell^t_{i+1}, \alpha^t_{i+1}, \ell^{t'}_0, \alpha^{t'}_0)$ be a thread creation operation where $\rho_i, \rho_{i+1} \in \mathit{Sh}$, $\ell^t_i, \ell^t_{i+1} \in \mathit{Lcl}(t)$, $\alpha^t_i, \alpha^t_{i+1} \in \mathit{Ctl}(t)$, $\ell^{t'}_0 \in \mathit{Lcl}(t')$ is the initial local store mapping for thread t', and $\alpha^t_0 \in \mathit{Ctl}(t')$ is the initial control state of t'. Let $v \in \mathcal{V}$ be a variable where $\ell^t_i(v) \neq \bot$ and $\rho_i(v) = \bot$ and assume that v becomes visible to t' during the thread creation operation. Then, $\ell^t_{i+1}(v) = \bot$ and $\rho_{i+1}(v) = \ell^t_i(v)$. Note that, the only influence of a thread creation operation on the shared variables is to change their visibility. Another effect of a thread creation operation is to change the set of threads within the program configuration. Let $tcop^t$ be a thread creation operation, c be the program configuration just before the execution of $tcop^t$, and c' be the program configuration just after the execution of $tcop^t$. Then, $c'(T) = c(T) \cup \{t'\}$. Also, c' contains the local store mappings and the control state of all the threads existing in c and the new thread t' ($\mathit{Lcl}(t')$ and $\mathit{Ctl}(t')$). The other type of operation that changes the set of threads in a program is the thread termination operation ($tdop^t$). If c and c' are the program configurations right before and after the execution of a termination operation $tdop^t$, respectively, then, $c'(T) = c(T) \setminus \{t\}$.

Environment interaction operations are the operations that a thread performs to communicate with its environment and that are not shared operations. The environment interaction operation types are 1) GUI operations, 2) RMI operations, 3) file read and write operations, 4) socket operations, and 5) command line argument read. We assume that the environment interaction operations do not affect the shared variables directly. A thread can reflect the effect of an environment interaction operation to the shared variables by executing shared operations after the execution of an environment interaction operation. An environment interaction operation performed by t is defined as $eop^t : \mathit{Lcl}(t) \times \mathit{Ctl}(t) \rightarrow \mathit{Lcl}(t) \times \mathit{Ctl}(t)$.

The other form of environment interaction is through input events. Input events are triggered outside of the program. For example, a button click event is an input event that is triggered by a user and delivered to the program. The input events can be GUI events (denoted as e_g) or the RMI events (denoted as e_r).

The GUI events are handled by the event thread t_e; therefore, a GUI event affects the configuration of t_e. Since the input events are triggered outside of the program, we assume that they do not affect the shared variables directly. The GUI events change the local store of t_e and t_e can later alter the shared variables by performing shared operations. An input event for a program is defined as $e_g : \mathit{Lcl}(t_e) \times \mathit{Ctl}(t_e) \rightarrow \mathit{Lcl}(t_e) \times \mathit{Ctl}(t_e)$.

Let us clarify our model for remote method invocation before we define the RMI events. Suppose a thread t of a program P invokes a remote method published by a program P'. This request is delivered to the runtime environment of P' and handled with the implicit RMI thread t_r that is responsible for this RMI connection. The remote method invocation by the thread t in P corresponds to an RMI operation (which is an environment interaction operation as discussed above). The request delivered to the RMI thread t_r in program P' corresponds to an RMI event. We define an RMI event captured by an RMI thread t_r as $e_r : \mathit{Lcl}(t_r) \times \mathit{Ctl}(t_r) \rightarrow \mathit{Lcl}(t_r) \times \mathit{Ctl}(t_r)$.

Program Execution: We will define the transitions in a program P with a relation $R_P : \mathcal{C} \times \mathcal{C}$ where $\mathcal{C}$ is the set of configurations of P. These transitions occur with an input event or with the execution of an operation by one of the threads. Let $(c, c') \in R_P$ be a transition of the program P with an operation op^t or event e. The configuration c is the program configuration just before op^t (or e), and the configuration c' is the program configuration just after the completion of op^t (or e). Below we define a number of transition relations for each operation and event type. Then, we give the definition of R based on these relations.

We introduce the following notations. Given a program configuration c, $c(\mathit{Sh}) \in \mathit{Sh}$ denotes the shared store mapping at the configuration c. Given a thread $t \in T$ and a program configuration c, $c(\mathit{Lcl}(t)) \in \mathit{Lcl}(t)$ denotes the local store mapping of t and $c(\mathit{Ctl}(t)) \in \mathit{Ctl}(t)$ denotes the control state of t at the configuration c.

We now define R_{sop^t}, the transition relation for the execution of a shared operation by a concurrent thread t. Recall that a shared operation is either a concurrency controller action execution or a shared data method invocation. When the operation is a shared data method invocation or a nonblocking action execution the transition relation is

$$\begin{aligned} R_{sop^t} = \ & \{(c,c')|c,c' \in \mathcal{C} \ \wedge \ c'(\mathit{Lcl}(t)) = c(\mathit{Lcl}(t)) \ \wedge \ sop^t(c(\mathit{Sh}), c(\mathit{Ctl}(t))) = (\rho, \alpha^t) \\ & \wedge\, c'(\mathit{Sh}) = \rho \ \wedge \ c'(\mathit{Ctl}(t)) = \alpha^t \ \wedge \\ & \wedge\, (\forall t' \in c(T),\ t' \neq t \Rightarrow c'(\mathit{Lcl}(t')) = c(\mathit{Lcl}(t')) \ \wedge \ c'(\mathit{Ctl}(t')) = c(\mathit{Ctl}(t')))\} \end{aligned}$$

If sop^t is a blocking action execution, then the transition relation is partitioned into two relations: $R^w_{sop^t}$ and $R^x_{sop^t}$ where $R^w_{sop^t}$ represents the case that all the guards of the blocking action evaluate to false and therefore the thread t is suspended, and $R^x_{sop^t}$ represents the case that one of the guards of the blocking action evaluate to true and therefore thread t executes the action. We define $R^w_{sop^t}$ as:

$$\begin{aligned} R^w_{sop^t} = \ & \{(c,c_w)|c,c_w \in \mathcal{C} \ \wedge \ c_w(\mathit{Sh}) = c(\mathit{Sh}) \\ & \wedge\, \forall gc \in a.GC, \neg gc.g(\textstyle\prod_{\gamma\in\Gamma} c(\mathit{Sh})(\gamma)) \ \wedge \ c_w(\mathit{Ctl}(t)) = w \ \wedge \ c_w(\mathit{Lcl}(t)) = c(\mathit{Lcl}(t)) \\ & \wedge\, (\forall t' \in c(T),\ t' \neq t \Rightarrow c'(\mathit{Lcl}(t')) = c(\mathit{Lcl}(t')) \ \wedge \ c'(\mathit{Ctl}(t')) = c(\mathit{Ctl}(t')))\} \end{aligned}$$

where a is the blocking action that suspends the thread t, w is the control point where the thread t starts to wait, and c_w is the program configuration when t is blocked. The transition relation $R^x_{sop^t}$ is defined as:

$$\begin{aligned} R^x_{sop^t} = \ & \{(c,c')|c,c' \in \mathcal{C} \\ & \wedge\, sop^t(c(\mathit{Sh}), c(\mathit{Ctl}(t))) = (\rho, \alpha) \ \wedge \ c'(\mathit{Sh}) = \rho \\ & \wedge\, \exists gc \in a.GC, gc.g(\textstyle\prod_{\gamma\in\Gamma} c(\mathit{Sh})(\gamma)) \ \wedge \ c'(\mathit{Ctl}(t)) = \alpha \ \wedge \ c'(\mathit{Lcl}(t)) = c(\mathit{Lcl}(t)) \\ & \wedge\, (\forall t' \in c(T),\ t' \neq t \Rightarrow c'(\mathit{Lcl}(t')) = c(\mathit{Lcl}(t')) \ \wedge \ c'(\mathit{Ctl}(t')) = c(\mathit{Ctl}(t')))\} \end{aligned}$$

Then, $R_{sop^t} = R^w_{sop^t} \cup R^x_{sop^t}$

We define R_{lop^t}, the transition relation for the execution of a local operation by a concurrent thread t as

$$\begin{aligned} R_{lop^t} = \ & \{(c,c')|c,c' \in \mathcal{C} \ \wedge \ c'(\mathit{Sh}) = c(\mathit{Sh}) \ \wedge \ lop^t(c(\mathit{Lcl}(t)), c(\mathit{Ctl}(t))) = (\ell^t, \alpha^t) \\ & \wedge\, c'(\mathit{Ctl}(t)) = \alpha^t \ \wedge \ c'(\mathit{Lcl}(t)) = \ell^t \\ & \wedge\, (\forall t' \in c(T),\ t' \neq t \Rightarrow c'(\mathit{Lcl}(t')) = c(\mathit{Lcl}(t')) \ \wedge \ c'(\mathit{Ctl}(t')) = c(\mathit{Ctl}(t')))\} \end{aligned}$$

Recall that, local operations do not affect the shared variables. Therefore, modeling a local operation execution as an atomic execution does not influence the correctness of the model with respect to synchronization behavior.

We define R_{eop^t}, the transition relation for the execution of an environment interaction operation by a concurrent thread t as

$$\begin{aligned} R_{eop^t} = \ & \{(c,c')|c,c' \in \mathcal{C} \ \wedge \ c'(\mathit{Sh}) = c(\mathit{Sh}) \ \wedge \ eop^t(c(\mathit{Lcl}(t)), c(\mathit{Ctl}(t))) = (\ell^t, \alpha^t) \\ & \wedge\, c'(\mathit{Ctl}(t)) = \alpha^t \ \wedge \ c'(\mathit{Lcl}(t)) = \ell^t \\ & \wedge\, (\forall t' \in c(T),\ t' \neq t \Rightarrow c'(\mathit{Lcl}(t')) = c(\mathit{Lcl}(t')) \ \wedge \ c'(\mathit{Ctl}(t')) = c(\mathit{Ctl}(t')))\} \end{aligned}$$

For thread creation operation we define the transition relation R_{tcop^t}, where t is the thread performing the operation $tcop$ and t'' is the thread created with that operation, as

$$\begin{aligned} R_{tcop^t} = \ & \{(c,c')|c,c' \in \mathcal{C} \ \wedge \ tcop^t(c(\mathit{Sh}), c(\mathit{Lcl}(t)), c(\mathit{Ctl}(t))) = (\rho, \ell^t, \alpha^t, \ell_0^{t''}, \alpha_0^{t''}) \ \wedge \ c'(\mathit{Sh}) = \rho \\ & \wedge\, c'(\mathit{Lcl}(t)) = \ell^t \ \wedge \ c'(\mathit{Ctl}(t)) = \alpha^t \ \wedge \ c'(\mathit{Lcl}(t'')) = \ell_0^{t''} \ \wedge \ c'(\mathit{Ctl}(t'')) = \alpha_0^{t''} \ \wedge \ c'(T) = c(T) \cup \{t''\} \\ & \wedge\, (\forall t' \in c(T),\ t' \neq t \ \Rightarrow c'(\mathit{Lcl}(t')) = c(\mathit{Lcl}(t')) \ \wedge \ c'(\mathit{Ctl}(t')) = c(\mathit{Ctl}(t')))\} \end{aligned}$$

We define R_{tdop^t}, the transition relation for termination operation as

$$\begin{aligned} R_{tdop^t} = \ & \{(c,c')|c,c' \in \mathcal{C} \ \wedge \ c'(T) = c(T) \setminus \{t\} \ \wedge \ tdop^t(c(\mathit{Sh}), c(\mathit{Lcl}(t)), c(\mathit{Ctl}(t))) = \rho \ \wedge \ c'(\mathit{Sh}) = \rho \\ & \wedge\, (\forall t' \in c(T),\ t' \neq t \Rightarrow c'(\mathit{Lcl}(t')) = c(\mathit{Lcl}(t')) \ \wedge \ c'(\mathit{Ctl}(t')) = c(\mathit{Ctl}(t')))\} \end{aligned}$$

We define a transition relation R_{e_g} which corresponds to the execution of the GUI event e_g as follows:

$$\begin{aligned} R_{e_g} = \ & \{(c,c')|c,c' \in \mathcal{C} \ \wedge \ c'(\mathit{Sh}) = c(\mathit{Sh}) \ \wedge \ e_g(c(\mathit{Lcl}(t_e)), c(\mathit{Ctl}(t_e))) = (\ell^{t_e}, \alpha^{t_e}) \\ & \wedge\, c'(\mathit{Lcl}(t_e)) = \ell^{t_e} \ \wedge \ c'(\mathit{Ctl}(t_e)) = \alpha^{t_e} \\ & \wedge\, (\forall t \in c(T),\ t \neq t_e \Rightarrow c'(\mathit{Lcl}(t)) = c(\mathit{Lcl}(t)) \ \wedge \ c'(\mathit{Ctl}(t)) = c(\mathit{Ctl}(t)))\} \end{aligned}$$

The transition relation for an RMI event e_r captured with an RMI thread t_r is defined as:

$$\begin{aligned} R_{e_r} = \ & \{(c,c') \mid c, c' \in \mathcal{C} c'(Sh) = c(Sh) \ \wedge \ e_r(c(Lcl(t_r)), c(Ctl(t_r))) = (\ell^{t_r}, \alpha^{t_r}) \\ & \wedge c'(Lcl(t_r)) = \ell^{t_r} \ \wedge \ c'(Ctl(t_r)) = \alpha^{t_r} \\ & \wedge (\forall t' \in c(T), t' \neq t_r \Rightarrow c'(Lcl(t')) = c(Lcl(t')) \ \wedge \ c'(Ctl(t')) = c(Ctl(t'))\} \end{aligned}$$

Finally, the transition relation R^P is the union of all of the event and operation execution transition relations for all of the input events in the program and all of the operations performed by all of the threads. $R^P = \bigcup_{op^t \in OP^t, t \in T} R_{op^t} \cup \bigcup_{e \in E} R_e$ where T is the set of threads, E is the set of events, and OP^t is the set of operations that a thread t can perform.

An execution of a program P is defined as follows. P starts with the initial configuration $c_0 = (\rho_0, \ell_0^{t_m}, \alpha_0^{t_m}, \ell_0^{t_e}, \alpha_0^{t_e}, \ell_0^{t_{r0}}, \alpha_0^{t_{r1}}, \ell_0^{t_{r1}}, \alpha_0^{t_{r2}}, \ldots, \ell_0^{t_{rk}}, \alpha_0^{t_{rk}})$. The program configuration is updated with input events and operations according to the transition relation R_P defined above. We represent one step program execution with an input event e as $c \xrightarrow{e} c'$ where $(c, c) \in R_e$. Given an operation op which is not a blocking action execution, we denote the one step program execution which corresponds to execution of op by thread t as $c \xrightarrow{op^t} c'$ where $(c, c') \in R_{op^t}$. For a blocking action execution, we denote the one step program execution which corresponds to execution of op by thread t as $c \xrightarrow{wop^t} c'$ if $(c, c') \in R^w_{op^t}$ and $c \xrightarrow{op^t} c'$ if $(c, c') \in R^x_{op^t}$. A program execution $x_p = x_0, x_1, \ldots, x_i, x_{i+1}, \ldots$ is a sequence where the first element x_0 is a one step execution starting from the initial configuration c_0, and each element x_i in this sequence is of the form $x_i = c_i \xrightarrow{label} c_{i+1}$ with $(c_i, c_{i+1}) \in R^P$ and $label$ is an operation or input event.

3.3.2. Correctness Criteria for Interface and Behavior Verification

In this section we define two projection functions. We use these projection functions in defining the correctness criteria for interface and behavior verification.

First we introduce a product interface machine I^p for concurrency controller interfaces. Given k controller interfaces $I_1, I_2, \ldots, I_k$ of k concurrency controllers where $I_i = (Q_i, q_{0_i}, \Sigma_i, \delta_i, F_i)$ and $1 \leq i \leq k$, we define the product machine as $I^p = (Q^p, q_0^p, \Sigma^p, \delta^p, F^p)$. The set of states in I^p is $Q^p = Q_1 \times Q_2 \times \cdots \times Q_k$. The initial is $q_0^p = (q_{0_1}, q_{0_2}, \ldots, q_{0_k})$. The input alphabet is $\Sigma^p = \bigcup_{1 \leq i \leq k} \Sigma_i$ and the set of final states is $F^p = \{q_0^p\}$. The transition function δ^p is defined as follows. Given a state $q \in Q^p$ and an integer $1 \leq i \leq k$, let $q[i] \in Q_i$ denote the ith element of q. Given a state $q \in Q^p$ and an input symbol $\sigma \in \Sigma$, the next state is $q' = \delta^p(q, \sigma)$ if and only if $q' \in Q^p$ and $\exists 1 \leq i \leq k, \ s.t. \ \delta_i(q[i], \sigma) = q'[i] \ \wedge \ (\forall 1 \leq j \leq k, j \neq i \Rightarrow q[j] = q'[j] = q_{0_j})$. Note that, this product machine encodes the third assumption stated at the beginning of Section 3.3.1, i.e., a thread executes an action of a controller only if it is at the initial interface state with respect to all other controllers.

Here we introduce our first projection function $\Pi_1 : X_p \times T \to SOp$ where X_p is the set of all program executions, T is the set of threads, and SOp is a set of shared operation sequences. Given a program execution $x_p = x_0, x_1, \ldots$ where $x_p \in X_p$ and a thread t, the function $\Pi_1(x_p)(t)$ removes from x_p the configurations, input events, and the operations other than the shared operations performed by t. The projection $\Pi_1(x_p)(t)$ is defined recursively as follows. Let c_j be the first program configuration from which t performs a shared operation. I.e., the shared operation at $c_j \xrightarrow{sop_0^t} c_{j+1}$ is the first shared operation performed by t. The projection $\Pi_1(x_0, \ldots, x_{j-1})$ is the empty sequence. The projection $\Pi_1(x_0, \ldots, x_j)$ is sop_0^t. The projection $\Pi_1(x_0, \ldots, x_j, \ldots, x_{l+1})$ for $l > j$ is $\Pi_1(x_0, \ldots, x_j, \ldots, x_l), sop_z^t$ if x_{l+1} is $c_{l+1} \xrightarrow{sop_z^t} c_{l+2}$. Otherwise, the projection is $\Pi_1(x_0, \ldots, x_j, \ldots, x_{l+1}) = \Pi_1(x_0, \ldots, x_j, \ldots, x_l)$.

Recall that, based on the design for verification approach, the shared operations are known and explicit in the program. Moreover, these operations are either controller actions or shared data operations according to the concurrency controller pattern. Therefore, the result of this projection function is a sequence of controller actions and shared data operations, listed in the order they are performed by t.

Using the product machine definition I^p and the projection Π_1 we give the following definition to be used as the correctness criteria during the interface verification.

Definition 3.1. (Thread Interface Correctness) Let $I_0, \ldots I_k$ be the interfaces of the concurrency controllers used by a thread t and I^p be the product machine of these interfaces. The thread t is **interface correct** if for all $x_p \in X_p$, $\Pi_1(x_p)(t)$ is a legal sequence of the product machine I^p.

Before introducing the projection function to be used for behavior verification, we define the obedience of a thread to the interface of a controller. First we extend the definition of Π_1 with a concurrency interface attribute. Given a program execution sequence x_p, an interface I, and a thread t, $\Pi_1(x_p)(t)(I)$ returns the sequence of input symbols of I in the order they appear in x_p. The recursive definition of this projection is similar to the one above. Using this projection function, we define thread obedience as follows.

Definition 3.2. (Thread Obedience) Given a thread t and a controller interface I, t **obeys** I if for all execution sequences $x_p \in X_p$, $\Pi_1(x_p)(t)(I)$ is a legal sequence of I.

Note that, thread obedience is weaker than the thread correctness since it considers only one controller interface.

Now we define the second projection function Π_2. Given a program execution $x_p = x_0, x_1, \ldots$ where $x_p \in X_p$ and a concurrency controller $CC = (\Gamma, IC, A, I)$, the function $\Pi_2(x_p)(CC)$ computes a projection of x_p on the concurrency controller CC. Let n be the maximum number of threads in a program execution $x_p \in X_p$. Let c_j be the program configuration that has the initial state of the controller CC. (I.e., c_j is the first configuration where there exists a store mapping that maps the concurrency controller to a value other then $\perp$ or uninitialized.) The projection $\Pi_2(x_p)(CC)$ is computed recursively as follows. $\Pi_2(x_0, ..., x_{j-1})(CC)$ is the empty sequence. $\Pi_2(x_0, \ldots, x_j)(CC)$= $\prod_{\gamma\in\Gamma} \ell_j^t(\gamma) \times \prod^n q_0$ where q_0 is the initial state of I and t is the thread that initializes the controller. The projection $\Pi_2(x_0, \ldots, x_j, \ldots, x_{l+1})(CC)$ is $\Pi_2(x_0, \ldots, x_j, \ldots, x_l)(CC)$ if x_{l+1} is not $c_{l+1} \xrightarrow{sop^t} c_{l+2}$ for some $t \in T$ where *sop* is an action of CC. Otherwise, it is computed, assuming that the threads obey I, as follows. Let $a \in A$ be the action for the operation sop^t. For each concurrent thread $t_z \in T$ and $t_z \neq t$, let q_{t_z} be the interface state of t_z at the last element of the sequence $\Pi_2(x_0, \ldots, x_j, \ldots, x_l)(CC)$, and let q_t be that of t. The next interface state for each thread t_z other than t is $q'_{t_z} = q_{t_z}$. The next interface state (q'_t) of t is $q'_t = \delta(q_t, a)$. Finally, $\Pi_2(x_0, \ldots, x_j, \ldots, x_{l+1})(CC)$ is $\Pi_2(x_0, \ldots, x_j, \ldots, x_l)(CC)$, $(\prod_{\gamma\in\Gamma} \rho_{l+1}(\gamma) \times q'_t \times \prod_{t\in T\ \wedge t_z\neq t} q'_{t_z})$.

The condition that all threads obey I is required since the computation of Π_2 relies on this assumption to construct such a sequence. If one of the threads violates this condition, the projection computation cannot find a next interface state for that thread at the violation point and cannot construct such a sequence.

Theorem 3.3. Let X_p be the set of all executions of P. Given a concurrency controller $CC = (\Gamma, IC, A, I)$, if all the threads that use CC obey $I = (Q, q_0, \Sigma, \delta, F)$ then $\bigcup_{x_p\in X_p}\{\Pi_2(x_p)(CC)\}$ is the subset of the set of execution paths in $T(CC)(n)$, where n is the maximum number of threads in all executions in X_p.

Proof. To prove this theorem, we need to show that if all the threads using CC obey I based on Definition 3.2, then 1) each element of the sequence produced by $\Pi_2(x_p)(CC)$ is an element of ST, and 2) for any two consecutive elements of the sequence $\Pi_2(x_p)(CC)$, say p_i and p_{i+1}, there is a tuple $(s_i, s_{i+1}) \in RT$.

The first condition holds trivially by the definition of the projection function since each element of the projection is in $\prod_{\gamma\in\Gamma} \text{DOM}(\gamma) \times \prod^n Q$ which is the definition of ST. Note that, the first element of the sequence resulting from $\Pi_2(x_p)(CC)$ is an element of IT due to the projection function definition. Also, in the projection result, the uncreated threads are at initial interface state, which is the case in IT, and the interface state of terminated threads do not change.

Now we examine the second condition. Let p_i, p_{i+1} be two consecutive elements of $\Pi_2(x_p)(CC)$. Consider the controller variables in p_i and p_{i+1}. Let sop^t be the shared operation performed by some thread t that is used during the computation of p_{i+1} and let ρ_i, ρ_{i+1} be the shared store mappings before and after the operation sop^t, respectively. According to the definition of projection function, *sop* is an action of CC. In the following discussion, we denote this action as a. Let q_i^t be the interface state of t in p_i and q_{i+1}^t be the interface state of t in p_{i+1}. According to the definition of the projection function, the interface states of all other threads are the same in both p_i and p_{i+1}. If t obeys I, then according to Definition 3.1 there is an interface state $\delta(q_i^t, a) = q_{i+1}^t$. Moreover, there is a tuple $(s_i, s_{i+1}) \in RT$ where $s_{i+1}(\gamma) = \rho_{i+1}(\gamma)$ and $s_i(\gamma) = \rho_i(\gamma)$ for all $\gamma \in \Gamma$, $s_{i+1}(Q)(t) = q_{i+1}^t$, $s_i(Q)(t) = q_i^t$, and $s_{i+1}(Q-t) = s_i(Q-t)$. Therefore, $(p_i, p_{i+1}) \in RT$ if t obeys I. □

Theorem 3.3 leads to the following.

Corollary 3.4. Given a concurrency controller $CC = (\Gamma, IC, A, I)$ and a program P that has an instance of CC accessed by n threads, the ACTL properties verified on transition system $T(CC)(n)$ for the concurrency controller CC are preserved in P if all the threads are interface correct.

Proof. To show that the ACTL properties of $T(CC)(n)$ are preserved in the program P we need to show that $T(CC)(n)$ simulates P [15]. For this purpose we define a simulation relation $H \subseteq \mathcal{C} \times ST$ where $\mathcal{C}$ is the configuration set of P and ST is the state set of $T(CC)(n)$. This relation is as follows. $H = \{(c, s) \mid c \in \mathcal{C}\ \wedge s \in ST\ \wedge\ (\exists x_p = x_0, x_1, \ldots, x_i, \cdots \in X_p, x_i = c_i \xrightarrow{label} c\ \wedge\ \Pi_2(x_0, x_1, \ldots, x_i)(CC) = s_0, s_1, \ldots, s)\}$ where X_p is the set of executions of P. Such a simulation function exists if all the threads are interface correct since the set of projection Π_2 results for any $x_p \in X_p$ is a subset of the transition system $T(CC)(n)$ according to Theorem 3.3. Therefore, the ACTL properties of $T(CC)(n)$ are preserved in the program P. $\square$

Based on this corollary, the behavior verification of a concurrency controller is performed on the $T(CC)(n)$ in our framework.

4. Verification of Concurrency Controllers

In this section, we present our modular verification approach based on the concurrency controller pattern. The verification consists of two steps: 1) *Behavior verification:* Verification of the properties of the concurrency controller classes assuming that the threads obey the controller interfaces; 2) *Interface verification:* Verification of the threads that use the concurrency controllers to make sure that they access the methods of the controllers and the shared data in the order specified by the interfaces. I.e., during interface verification we check whether there are any interface violations. In this modular verification strategy the two verification steps are completely decoupled.

4.1. Behavior Verification

In our modular verification approach, during behavior verification it is assumed that there are no interface violations. Based on this assumption, interfaces are used as environment models. Using controller interfaces we can encapsulate the controller behavior and perform the verification on this encapsulated behavior separately. Since, interfaces allow isolation of the behavior, application of domain specific verification techniques (which may not be applicable or scalable in general) becomes feasible. These domain specific verification techniques may enable verification of stronger and more complex properties than that can be achieved by more generic techniques.

During behavior verification of concurrency controllers, the domain specific verification tool we use is the Action Language Verifier (ALV) [9, 41]. ALV is an infinite state symbolic model checker and can verify specifications with unbounded integer variables. ALV takes a specification in Action Language and a set of CTL formulas as input. ALV uses conservative approximation techniques such as widening and truncated fixpoint computations to conservatively verify or falsify infinite state specifications with respect to the given CTL formulas [9, 41].

To perform behavior verification, the concurrency controllers written based on the concurrency controller pattern are automatically translated into Action Language. The behavior verification is performed based on the concurrency controller semantics and the projections discussed in Section 3.

Recall that the behavior of a concurrency controller is specified using the instances of `Action` class in a guarded command style, and the required execution order of these actions are specified with the controller interface. Using this ordering and the definition of actions, each controller action is automatically translated to the atomic actions of the Action Language. Consider the BB-MUTEX controller discussed in Section 2. An implementation of this concurrency controller (`BBMutexController`) with three controller variables is given in Fig. 3. There are three concurrency controller actions and two transitions (`take` and `put`) from the shared buffer. The controller interface for BB-MUTEX is given in Fig. 5. The automatically generated Action Language specification for the BB-MUTEX is given below.

```
module main()
 boolean busy; integer count;
 parameterized integer size;
 module BBMutex()
  enumerated pc {IDLE,RPROD,PROD,RCONS,CONS};
  initial: count=0 and busy=false and pc=IDLE;
  consume_acquire: pc=IDLE and count>0 and !busy and count'=count-1 and busy' and pc=RCONS;
  produce_acquire: pc=IDLE and count<size and !busy and count'=count+1 and busy' and pc=RPROD;
  release_0: pc=PROD and busy and busy'=false and pc=IDLE;
  release_1: pc=CONS and busy and busy'=false and pc=IDLE;
  take: pc=RCONS and pc'=CONS;
  put: pc=RPROD and pc'=PROD;
  BBMutex: consume_acquire | produce_acquire | release_0 | release_1 | take | put;
 endmodule
 main: BBMutex()| BBMutex()| BBMutex() | BBMutex();
endmodule
```

A controller specification in Action Language consists of a main module and a submodule. Each instantiation of the submodule corresponds to a thread. The variables of the main module correspond to the variables of the concurrency controller. In the above example, the `main` module has a submodule called `BBMutex`. Submodule `BBMutex` models the threads and corresponds to a thread class in Java. Submodule `BBMutex` has one local enumerated variable. This enumerated variable (`pc`) keeps track of the thread state which is represented by a state of the controller interface. This is the only information we need to know about a thread to verify the controller implementation. A thread can only be in one of the interface states. Each instantiation of a module will create different instantiations of its local variables.

An atomic action in the Action Language defines one execution step using the current-state values for the variables (denoted as unprimed variables) and the next-state values (denoted as primed variables). Our tool translates the guard expression of an action to a formula on current-state values, and the update expression to a formula on current-state and next-state values. As an example, consider the action `consume_acquire` in the `BBMutexController` class. In the constructor, this action is defined with one guarded command. The guard expression of this guarded command is `(count>0 && !busy)`, and the update expression is `count = count - 1; busy = true;`. In the interface of the controller the transition associated with `consume_acquire` changes the thread's state from `IDLE` to `RCONS`. Using this information, the translator constructs the atomic action `consume_acquire` shown in the automatically generated Action Language specification above. In this example, the state of a thread is altered by the shared buffer as well. The operations `put` and `take` change the thread's state as shown in the controller interface. These state changes are translated to `put` and `take` actions shown in the above Action Language specification by inspecting the interface implementations `BufferStub` and `BBMutexStateMachine`.

Since Action Language Verifier can handle infinite state systems we are able to verify concurrency controllers with unbounded integer variables (such as `count`) or parameterized constants (such as `size`). When a specification has a parameterized constant it is verified for all possible values of the parameterized constant (for example, `size`). Currently, the variable types supported by the Action Language are integer, boolean, and enumerated (there is also some support for linked lists). Therefore, we have to restrict the controller variables to these types to verify them with ALV (we use static integers as enumerated variables in the controller implementations). On the other hand, since the controller variables only need to store the state information required for synchronization, these basic types have been sufficient for modeling concurrency controllers we have encountered so far.

4.1.1. Parameterized Verifi cation

We use an automated abstraction technique, called counting abstraction [18], to verify the behavior of a concurrency controller with respect to arbitrary number of threads. Implementation of counting abstraction for Action Language specifications is presented by Yavuz-Kahveci et al. [42]. The basic idea is to define an abstract transition system in which the local states of the threads (corresponding to the states of the controller interface) are abstracted away, but the number of threads in each interface state is counted by introducing a new integer variable for each interface state.

Given a concurrency controller $CC = (\Gamma, IC, A, I)$ with an interface $I = (Q, q_0, \Sigma, \delta, F)$, the parameterized transition system for that concurrency controller $T(CC)(p) = (IT_p,\ ST_p,\ RT_p)$, where p is a parameterized constant, is defined as follows: The set of states of the parameterized system is defined as $ST_p = \prod_{\gamma\in\Gamma} \text{DOM}(\gamma) \times \prod_{cv\in CV} \text{DOM}(cv) \wedge (\sum_{cv\in CV} cv = p)$, where CV is the set of integer variables introduced by the counting abstraction. Each $cv \in CV$ corresponds to an interface state $q \in Q$ and represents the number of threads in the interface state q. The initial states and the transition relation of the parameterized system can be defined using linear arithmetic constraints on these new variables [42]. Below is an excerpt from the parameterized Action Language specification generated for BB-MUTEX.

```
module main()
 boolean busy; integer count;
 parameterized integer size;
 parameterized integer numInstance;
 module BBMutex()
  integer IDLE,RPROD,PROD,RCONS,CONS;
  initial: count=0 and busy=false;
  initial: IDLE=numInstance and RPROD=0 and PROD=0 and RCONS=0 and CONS=0;
  restrict:numInstance>0;
  consume_acquire: IDLE>0 and count>0 and !busy and
                   count'=count-1 and busy' and RCONS'=RCONS+1 and IDLE'=IDLE-1;
  produce_acquire: IDLE>0 and count<size and !busy and
                   count'=count+1 and busy' and RPROD'=RPROD+1 and IDLE'=IDLE-1;
  release_0: PROD>0 and busy and busy'=false and IDLE'=IDLE+1 and PROD'=PROD-1;
  release_1: CONS>0 and busy and busy'=false and IDLE'=IDLE+1 and CONS'=CONS-1;
  take: RCONS>0 and CONS'=CONS+1 and RCONS=RCONS-1;
  put: RPROD>0 and PROD'=PROD+1 and RPROD'=RPROD-1;
  BBMutex: consume_acquire | produce_acquire | release_0 | release_1 | take | put;
 endmodule
 main: BBMutex();
endmodule
```

In this parameterized specification, the integer variable `IDLE` denotes the number of threads in the interface state `IDLE`. A parameterized integer constant, `numInstance`, denotes the number of threads. This parameterized constant is restricted to be positive. When the above 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).

4.2. Interface Verification

During interface verification, we check whether the assumption used during behavior verification is guaranteed by the threads that use the concurrency controllers, i.e., we check that each thread that uses a controller obeys the interface of that controller. In Section 3, we formalized the correctness criteria for this step. The threads should follow the action orderings defined by the interfaces of the concurrency controllers and should access the shared data protected by these controllers at the allowed interface states specified in the data stubs.

Specifying the interfaces uniformly, using finite state machines, enables us to handle interface verification using a uniform verification technique. Therefore, during interface verification, it is more suitable to use the generic verification techniques developed for the purpose of applying model checking directly to existing programming languages. We use the model checker Java PathFinder (JPF) [39] for this purpose.

JPF is an explicit and finite state model checker for Java. It enables the verification of arbitrary pure Java implementations without any restrictions on data types. JPF supports property specifications via assertions that are embedded into the source code. It exhaustively traverses all possible execution paths for assertion violations. If JPF finds an assertion violation during verification, it produces a counter-example which is a program trace leading to that violation. In addition, JPF provides two nondeterminism utilities: `Verify.random` and `Verify.randomBool`. These utilities are used in thread isolation as well as in the finite state machine implementations. At verification time, JPF systematically analyzes the program for every value created by these utilities, i.e., JPF searches the program's state space exhaustively.

The interface verification with JPF relies on the assertions embedded in the finite state machine implementation `StateMachine` and the assertions in the data stubs `SharedStub` (see Fig. 2). These classes provide the interface specifications for the controllers. During interface verification, we use the interfaces as stubs to abstract the controller behavior that is verified during behavior verification. The stub substitution is performed as follows. We replace `Controller` classes with `ControllerStateMachine` classes implementing the `ControllerInterface` Java interface and shared data classes with `SharedStub` classes implementing the `SharedInterface` Java interface. With this transformation, the concurrency controller actions are directed to the `transition` method of the `StateMachine` and the shared data accesses are directed to the shared stub methods. In addition, during the verification of the threads for interface violations there is no need to consider interleavings of different threads since we are only interested in the order of calls to the controller methods by each individual thread, and since the only interaction among different threads is through shared objects that are protected using the concurrency controllers. In other words, we can verify each thread in isolation.

JPF model checks the resulting program after the stub substitution. When an action of concurrency controller is invoked, the `transition` method of `StateMachine` asserts that this action execution is valid at the current interface state using the transitions defined by `ControllerStateMachine`. Recall the shared buffer example protected with BB-MUTEX controller. When a thread invokes the `produce_acquire` action, the state machine asserts that the thread is in the interface state `IDLE` as encoded in the `BBMutexStateMachine` in Fig. 6 (a). When a shared stub method is invoked, JPF checks if this access is allowed at the current interface state which is defined as an assertion within the `SharedStub` class. For example, when a thread of the concurrent buffer example invokes the `put` method, JPF checks whether the thread is in the interface state `RPROD` as encoded in the `BufferStub` in Fig. 6 (b). If there is an assertion violation, then there is an interface violation and JPF outputs the counter-example leading to that violation. A concurrent thread implementation is interface correct if JPF does not report any assertion violations.

During the interface verification with JPF, we achieve improvements in the efficiency due to the following reasons. The usage of controller interfaces instead of concurrency controllers reduces the state space dramatically since `ControllerStateMachine` classes do not contain the controller variables, locks and associated synchronization statements. Another factor is the usage of shared data stubs that do not contain actual data manipulation operations. Moreover, the use of controller interfaces and stubs with JPF's nondeterminism utilities abstracts the influences of the environment on the thread behavior. Therefore, we are able to perform interface verification on each thread separately. This isolation eliminates the need to consider all possible thread interleavings. Because of these factors, we achieve significant state space reductions leading to a significant improvement in the efficiency and scalability.

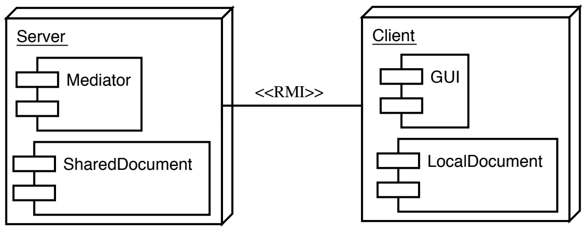
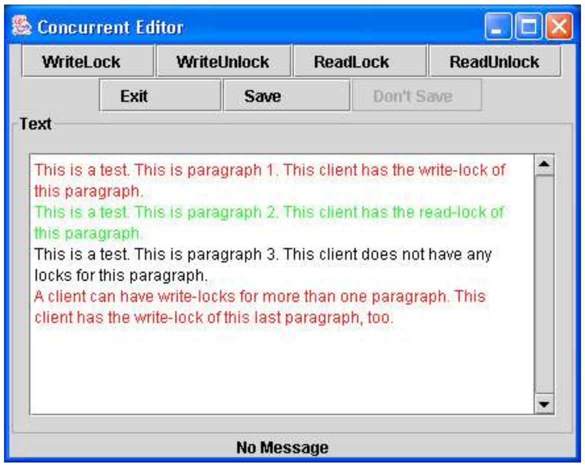


Fig. 8. Concurrent Editor architecture



This is a test. This is paragraph 1. This client has the write-lock of this paragraph.
This is a test. This is paragraph 2. This client has the read-lock of this paragraph.
This is a test. This is paragraph 3. This client does not have any locks for this paragraph.
A client can have write-locks for more than one paragraph. This client has the write-lock of this last paragraph, too.

No Message

Fig. 9. Concurrent Editor screen shot

5. Case Study and Experiments

In this section, we present the application of our design for verification approach with concurrency controllers to a concurrent text editor implementation, called Concurrent Editor. Concurrent Editor allows multiple users to edit a shared document at the same time as long as they are editing different paragraphs. Concurrent Editor is implemented as a distributed system that consists of a server node and a number of client nodes (one client node per user). Each node in this structure has its own copy of the shared document (Fig. 8).

Concurrent Editor maintains a consistent view of the shared document among the client nodes and the server. Users get write access to paragraphs of the document by clicking on the **WriteLock** button in the graphical user interface (GUI). Fig. 9 shows a screen shot. When **WriteLock** button is clicked, it generates a request for write access to the paragraph the cursor is on. When the request is granted, the color of the paragraph is changed to indicate that it is editable. A user can edit a paragraph only if she has the write access to that paragraph. Multiple users are able to edit different paragraphs of the document concurrently, i.e., each user is able to see the changes made by the other users as they occur. If a user wants to make sure that a paragraph does not change while reading it, she clicks the **ReadLock** button in the GUI. When a user has read access to a paragraph, other users can also have read access to that paragraph, but no other user can have write access to that paragraph. When a user has write access to a paragraph, no other user can have read or write access to that paragraph. All users have a copy of (and can see) the whole document, including the paragraphs they do not have read or write access. The GUI also provides **ReadUnlock** and **WriteUnlock** buttons to release the read and write accesses, respectively. Finally, the document is saved only when a consensus is reached among all client nodes.

5.1. Designing and Implementing the Concurrent Editor

When the above specification is given to a Java programmer, she has to consider both concurrent and remote accesses to the shared document. One can handle the remote access using the Java remote method invocation (RMI) and a

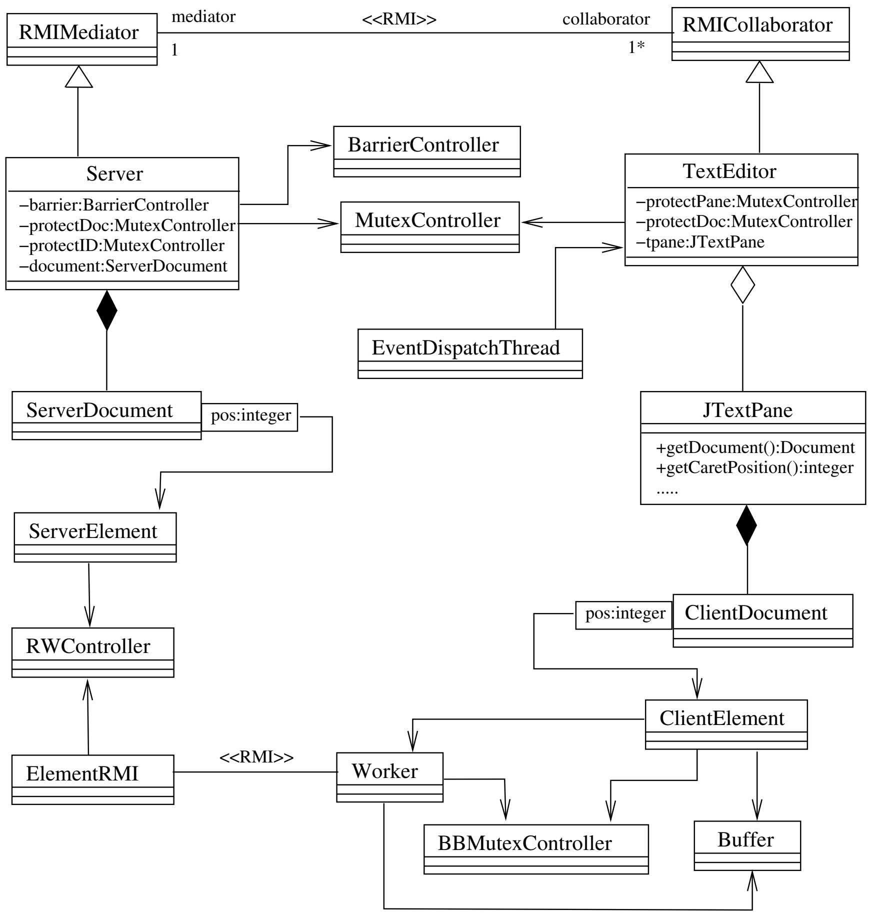


Fig. 10. Concurrent Editor class diagram

collaborative infrastructure where all client nodes register to a mediator through which the clients communicate [33] (see Fig. 8).

A number of synchronization issues need to be considered to handle the concurrent accesses to the shared document. Let us first focus on coordinating concurrent access to a single paragraph in the shared document. A naive solution is to declare all methods of the paragraph as `synchronized`, which is obviously not efficient. If mutual exclusion is enforced at every method, a client node requesting a read access will be blocked by another client node requesting the same access. The programmer needs to use a reader-writer lock (RW) to achieve a more efficient synchronization. A common methodology is encapsulating the synchronization policy within the shared data implementation, e.g., implementing the `write` method of the paragraph so that it acquires and releases the write lock implicitly. However, this methodology contradicts with the requirements of the Concurrent Editor, since it forces the write lock to be acquired at every write request. Therefore, the programmer needs to separate the lock acquisition from the actual write operation. A welcome side effect of this separation is that the programmer can change the paragraph implementation without affecting the synchronization policy. Similarly, one synchronization policy could be replaced with another without changing the paragraph implementation.

Another synchronization constraint that arises in this implementation is the following. While protecting the shared document, we cannot afford to suspend a client. Otherwise, the whole application will stall due to the collaborative infrastructure. To prevent the stalling of a client node, we need a separate worker thread to acquire the locks from the server node. This way, whenever the server node forces the requesting thread to wait, it does not suspend the client node. The use of a worker thread generates the need for a buffer for passing the requests between the client node and the worker threads, and the buffer needs to be protected by a synchronization policy. It is clear that the accesses to this buffer should be mutually exclusive. In this application, however, a mutual-exclusion (MUTEX) lock is not enough.

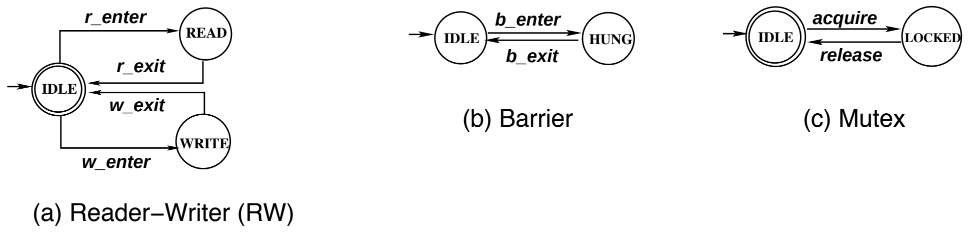


Fig. 11. Interfaces of the concurrency controllers used in the Concurrent Editor

We need a conditional wait for worker threads when the buffer is empty. We also need to put a bound to the buffer size to provide a reasonable response time to the user. Therefore, we protect this buffer by using a bounded buffer synchronized with a mutex lock (BB-MUTEX). In this synchronization policy, if a thread wants to take an item from the buffer it would wait while the buffer is empty. This policy also restricts the number of threads accessing the buffer, e.g., at any given time only one producer or consumer thread can access the buffer.

We have implemented the Concurrent Editor based on the concurrency controller pattern while considering the above constraints and issues. Fig. 10 shows the class diagram of our Concurrent Editor implementation using the concurrency controller pattern. The server node has a document of type `ServerDocument`. Mutually exclusive access to server document is ensured by a MUTEX controller. The server document consists of a number of paragraph elements. Each paragraph is associated with a unique reader-writer controller (RW) coordinating the read and write accesses to that paragraph. This node also has a barrier controller (BARRIER) which is used whenever a save request is issued by one of the users. A document is saved to disk only if a consensus is reached among all the users of the document.

A client node has a graphical user interface (GUI) with an editable text area and seven buttons (see Fig. 9). The text area is of type `JTextPane` containing a document of type `ClientDocument`. Both the text pane and the document are protected with MUTEX controllers. The text of the client document is a copy of the server document. There is an event thread (`EventDispatchThread`) running on the client side. In addition to the event thread, each paragraph element in the client document is associated with a unique worker thread created by the event thread. Each worker thread communicates with `ElementRMI` object of the corresponding paragraph in the server node via RMI calls. The `ElementRMI` object handles the remote accesses to the reader-writer controller of the corresponding paragraph in the server node. The communication between the event thread and the worker thread for a paragraph is via a communication buffer associated with that paragraph. The access to this buffer is controlled by a bounded-buffer mutex controller (BB-MUTEX). The interfaces of the concurrency controllers used both in client and server node of the Concurrent Editor implementation, other than the interface of BB-MUTEX, are shown in Fig. 11.

5.2. Behavior Verification Results

In order to verify the behavior of the concurrency controllers with ALV we need a list of properties to specify the correct behavior of the controllers. We allow the ACTL properties for the controllers to be either inserted directly to the generated Action Language specification or written as annotations in the controller classes (which are then automatically inserted into the Action Language translation).

The properties of the concurrency controllers used in the Concurrent Editor are shown in Table 1. The first row is the property for the BARRIER controller. The rows marked with PMUTEX1–12 shows the properties for the MUTEX controller. The properties PMUTEX1 and PMUTEX2 only refer to the controller variables. For example, the global property PMUTEX1 states that whenever `busy` is true, it must eventually be false. The remaining properties for this controller refer to both the controller variables and to the states of the threads. Note that the representation of the thread state is different in the concrete and the abstract Action Language specifications. The properties PMUTEX3–7 are for concrete specifications and refer to concrete thread states. For example, the property PMUTEX3 states that whenever a thread is in the `LOCKED` state it will eventually reach the `IDLE` state. The properties PMUTEX8–12 are for the parameterized instances with counting abstraction and refer to the integer variables `IDLE` and `LOCKED` which represent the number of threads in the interface state `IDLE` and `LOCKED`, respectively. For example, property PMUTEX8 states that whenever the number of threads that are in the `LOCKED` state is greater than zero, the value of the variable `busy` is false. There is a similar classification of the properties for the RW controller. The properties PRW1–3 only refer to the controller variables, which are the integer variable `nR` denoting the number of readers in the critical section and the boolean variable `busy` denoting if there is a writer in the critical section. For example, the global property RP1 states

Table 1. Properties of the concurrency controllers

PBARRIER	$\forall x\ AG((HUNG = x \wedge count < limit) \Rightarrow AX(HUNG \geq x))$
PMUTEX1	$AG(busy \Rightarrow AF(\neg busy))$
PMUTEX2	$AG(\neg busy \Rightarrow AF(busy))$
PMUTEX3	$AG(pc = LOCKED \Rightarrow AF(pc = IDLE))$
PMUTEX4	$AG(\neg(pc1 = LOCKED\ \wedge\ pc2 = LOCKED))$
PMUTEX5	$AG(pc = LOCKED \Rightarrow busy)$
PMUTEX6	$AG(\neg busy \Rightarrow pc = IDLE)$
PMUTEX7	$AG(\neg(pc1 = LOCKED\ \wedge\ pc2 = LOCKED))$
PMUTEX8	$AG(LOCKED > 0 \Rightarrow busy)$
PMUTEX9	$AG(\neg busy \Rightarrow IDLE > 0)$
PMUTEX10	$AG(LOCKED > 0 \Rightarrow AF(IDLE > 0))$
PMUTEX11	$AG(!(LOCKED > 2))$
PMUTEX12	$AG(LOCKED \geq 0 \wedge LOCKED \leq 1)$
PBB-MUTEX1	$\forall x\ AG((count = x \wedge pc \neq IDLE) \Rightarrow AX(count = x))$
PBB-MUTEX2	$AG(count \geq 0 \wedge count \leq size)$
PRW1	$AG(busy \Rightarrow nR = 0)$
PRW2	$AG(busy \Rightarrow AF(\neg busy))$
PRW3	$\forall x\ AG(nR = x \wedge nR > 0 \Rightarrow AF(nR \neq x))$
PRW4	$AG(pc = WRITE \Rightarrow AF(pc = IDLE))$
PRW5	$AG(\neg(pc1 = READ \wedge pc2 = WRITE))$
PRW6	$AG(\neg(pc1 = WRITE \wedge pc2 = WRITE))$
PRW7	$AG(pc1 = READ \Rightarrow nR > 0)$
PRW8	$AG(pc1 = WRITE \Rightarrow busy)$
PRW9	$AG(WRITE > 0 \Rightarrow AF(WRITE = 0))$
PRW10	$AG(\neg(READ > 0 \wedge WRITE > 0))$
PRW11	$AG(\neg(WRITE > 1))$
PRW12	$AG(WRITE = 1 \Leftrightarrow busy)$
PRW13	$\forall x\ AG(READ = x \wedge READ > 0 \Rightarrow AF(READ \neq x))$
PRW14	$AG(READ = nR)$

Table 2. Behavior verification performance for concrete instances with 8 threads and for parameterized instances

Instance	Time (s)	Memory(MB)	P-Time(s)	P-Memory(MB)
BARRIER	0.07	1.45	0.01	0.51
MUTEX	0.02	0.41	0.03	0.98
BB-MUTEX	1.65	19.35	0.27	1.46
RW	2.26	6.33	8.10	12.05

that whenever `busy` is true `nR` must be zero. The properties PRW4–8 are for concrete specifications and refer to concrete thread states. For example, the property RP4 states that whenever a thread is in the WRITE state it will eventually reach the IDLE state. The properties PRW9–14 are for the parameterized instances and refer to the integer variables which represent the number of threads in a particular state. For example property PRW14 states that at any time the number of threads that are in the reading state is the same as the value of the variable `nR`. The properties for the BB-MUTEX controller are marked as PBB-MUTEX1 and PBB-MUTEX2. Note that, the controller BB-MUTEX should also satisfy the properties of the MUTEX (PMUTEX1–12). Note that, five of the properties shown in Table 1 contain universally quantified integer variables. We are able to check such properties with ALV by declaring the universally quantified variables as parameterized constants.

We applied the presented behavior verification technique with ALV to the concurrency controllers used in the Concurrent Editor. Table 2 shows the performance of ALV for two generated instances of each of these concurrency controllers: one concrete instance with 8 threads and one parameterized instance using counting abstraction. The verification results for the parameterized instances are stronger compared to the concrete cases since they indicate that the verified properties hold for arbitrary number of threads. We verified all of the associated properties given in Table 1 with ALV for each concrete and parameterized concurrency controller instance. In this table, the first two columns (Time and Memory) show the time and memory consumed for the verification of the concrete instances with 8 threads and the last two columns (P-Time and P-Memory) show the time and memory consumed for the verification of the parameterized instances.

Table 3. Interface verification performance

Node-Thread	Time(s)	Memory(MB)
Server-Main	2.77	3.20
Server-RMI	185.85	67.14
Server-ElementRMI	3.28	2.79
Client-Main	2.92	3.87
Client-RMI	229.18	127.94
Client-Event	1636.62	139.48
Client-Worker	5.13	8.02

For these concurrency controllers, the behavior verification took a small amount of time and used a fraction of the memory. This is true even for the parameterized instances.

5.3. Interface Verifi cation Results

We have performed interface verification to ensure that there are no interface violations in the implementation of the Concurrent Editor. First we have identified the threads in the implementation. Recall that, the Concurrent Editor is comprised of a client node and a server node. The client node of Concurrent Editor contains a main thread, two implicit threads (event thread and RMI thread), and explicitly created worker threads. The server node contains a main thread, one RMI thread per client, and another RMI thread per paragraph serving the remote calls initiated by the worker threads. We performed interface verification on each thread separately. Table 3 shows the JPF performance during the interface verification for each of these threads. The server threads are labeled with Server and the client threads are labeled with Client.

In the client node, the main thread does not modify any shared data. This thread only instantiates the client application and opens an RMI connection, which is an environment interaction operation (*eop*) and modeled with stubs by using JPF's nondeterminism utilities. The event thread interacts with its environment through shared operations, which are modeled by interfaces, and a number of input events, which are modeled within drivers. A driver is a simple program with a main function that creates, in a loop, input events selected nondeterministically via JPF's utilities. During our experiments, since the client has a large number of GUI components and JPF exhausts memory for all possible event lengths, we used a driver which only generates all possible GUI event sequences of length 2. The other implicit thread, the RMI thread, is responsible for serving 5 different kinds of RMI input events. The rest of the threads in the client node are the worker threads. Note that these threads share the same class definition. Therefore, we can perform interface verification on one worker instance and generalize the result for all worker threads since we reflect the influences of other threads with controller interfaces and thread creation model. In the server node, the main thread does not modify any shared data as well. The RMI threads for serving the client nodes interact with their environment through 7 kinds of RMI input events and shared operations. The RMI threads for paragraphs interact with their environment through 4 kinds of RMI input events.

During the interface verification of Concurrent Editor we have discovered several interface violation errors. One group of these errors was caused by not calling the correct controller method before accessing the shared data. Another group of errors was a violation of the controller call sequence because of the incorrectly handled exception blocks. Our experience shows the necessity of the interface verification and also shows that such errors can be captured in a realistic system such as the Concurrent Editor. In [6] this verification approach is applied to verification of a safety critical air traffic control software and the results obtained there also confirm this view.

To demonstrate the effectiveness of our modular verification approach we conducted another experiment. We verified the concurrency controllers used in the case study with JPF without using controller interfaces as stubs. Table 4 shows the JPF performance for these controllers. For this experiment, the user threads are kept simple. The threads execute concurrency controller actions and invoke the shared stub methods in an infinite loop. There are no other computations. In this experiment, all the threads obeyed the controller interfaces. We used both depth-first and breadth-first search heuristics of JPF. The column labeled TN-S shows the number of user threads and the buffer size (for BB-MUTEX). For the rest of the controllers this column shows the number of user threads only. The memory usage is shown in the column labeled M, and the execution time is displayed in the column labeled T. For the BB-MUTEX case it is not possible to verify the original specification using a program checker such as JPF since the size of the buffer is an unspecified constant. To evaluate the performance of JPF we picked a fixed buffer size in the experiments reported in Table 4. Although the memory usage is low, the time spent grew exponentially because of the state space

Table 4. Performance of JPF without using interfaces as stubs

		DFS		BFS	
Instance	TN-S	T(s)	M(MB)	T(s)	M(MB)
BB-MUTEX	2 – 2	33.96	5.77	39.59	6.78
BB-MUTEX	2 – 3	42.66	9.06	57.58	5.65
BB-MUTEX	2 – 4	63.99	9.81	84.49	6.69
BB-MUTEX	2 – 5	119.02	16.38	121.34	8.85
BB-MUTEX	2 – 6	115.24	14.59	152.64	20.55
BB-MUTEX	2 – 7	151.17	25.00	199.60	20.18
BB-MUTEX	3 – 2	1133.76	35.95	↑	↑
RW	2	10.88	6.64	13.42	4.74
RW	3	262.20	16.05	332.24	16.49
RW	4	8017.89	172.92	13298.37	212.19
RW	5	↑	↑	↑	↑
MUTEX	2	18.05	4.79	8.43	5.30
MUTEX	3	163.74	8.75	145.14	30.80
MUTEX	4	4316.94	86.28	4187.76	97.88
MUTEX	5	↑	↑	↑	↑
BARRIER	2	12.34	4.4	13.75	2.8
BARRIER	3	95.93	5.51	72.62	9.68
BARRIER	4	1852.92	31.10	1511.28	60.10
BARRIER	5	35364.32	500.61	↑	↑

explosion. The memory usage is low since there is only one item instance and the same instance is inserted into the buffer numerous times in this experiment. If there were different item instances inserted to the buffer, the memory consumption would be much greater. The table also shows the performance when we increase the number of threads. For the BB-MUTEX case, when there are 3 user threads with buffer size 3, JPF spent more than 2 hours for both heuristics (denoted by ↑). For the rest of the concurrency controllers the performance of JPF dropped dramatically when the number of user threads increased because of the increase in the number of possible interleavings. JPF cannot handle these controllers when there are 5 user threads as shown in the table. On the other hand, the presented interface verification for the very same threads was performed successfully without exhausting the memory. The threads of BB-MUTEX were verified in 6.21 seconds and used 8.92 MB memory. The threads of RW were verified in 6.21 seconds and used 8.92 MB memory. The threads of BARRIER were verified in 2.80 seconds and used 3.25 MB memory. The threads of MUTEX were verified in 2.90 seconds and used 3.52 MB memory. When using controller interfaces as stubs, the reachable state spaces become finite and we do not have to consider all possible thread interleavings. Therefore, we have achieved a significant state space reduction; thus, a dramatic improvement in the efficiency of the interface verification.

6. Related Work

Earlier work on design for verification focused on verification of UML models [36] and use of design patterns in improving the efficiency of automated verification techniques [32]. Sharygina et al. [36] focus on verification of UML models, whereas we focus on verification of programs. Similar to our work, Mehlitz et al. [32] also suggest using design patterns in improving the efficiency of the automated verification. Our interface-based modular verification technique, however, is different and does not overlap with their approach [32, 31]. ESC Java [21] uses an approach based on design by contract and automated theorem proving which is similar to what we are proposing in this paper for model checking.

There has been other work on stateful interfaces. In [13] interfaces of software modules are specified as a set of constraints, and algorithms for interface compatibility checking are developed. DeLine et al. [16, 17] extend type systems with stateful interfaces and suggest an approach in which interface checking is treated as a part of type checking. Interface discovery and synthesis in which stateful interfaces are extracted by analyzing existing code are also studied [40, 1]. We use finite state machines to specify interfaces as a part of a design pattern and verify both the controller behavior and conformance to interface specifications.

Assume guarantee style verification of software components has also been studied [34] in which LTL formulas are used to specify the environment (i.e., the interface) of a component. Unlike specifying the interface with LTL formulas, in our approach, we use finite state machines for specifying interfaces and apply an interface-based assume-guarantee style verification. Automated environment generation for software components has been investigated using techniques such as inserting nondeterminism into the code and eliminating or restricting the input arguments by using side effect and points-to analyses [23, 37, 38].

Similar to our modularity in the interface verification, Flanagan et al. [22] presents a thread-modular reasoning and verifies each thread separately with respect to safety properties. The effects of other threads are modeled as environment assumptions whereas we use stubs and drivers to reflect these effects. Besides, we check the thread behavior against the interface rules and leave the assurance of the safety properties to behavior verification.

Design patterns for multi-threaded systems have been studied extensively. For example, Schmidt et al. [35] present several interrelated patterns, including synchronization and concurrency patterns, for building concurrent and network systems. Some of these patterns, such as Active Object, Monitor Object and Strategized locking pattern, are closely related to our concurrency controller pattern. Lea [29] also discusses several design patterns for concurrent object oriented programming and their usage in Java programs. All these patterns are built to help developers in writing reliable concurrent programs. Our goal, on the other hand, is to present a design pattern which improves the verifiability of concurrent programs by automated tools. In addition to introducing a verifiable design pattern for concurrency, we also propose a modular verification technique that exploits the presented design pattern.

Lea [29] also provides a package of Java solutions for commonly used synchronization policies. Our concurrency controller implementations could be interpreted as a generalization of this framework. Our framework enables customized solutions for customized synchronization policies. A developer can write her own synchronization policy without much effort when she faces a new problem which requires a customized solution.

Deng et al. [19] propose a pattern system. The patterns in their system are idioms that are used for specifying a synchronization policy in a high-level language. These specifications are also used as abstractions when extracting the model of the program with the synthesized code to reduce the cost of automated verification. In our approach, we achieve the state space reduction during interface checking by replacing the controller implementations with the controller interfaces that serve as stubs. Although these approaches may seem similar, there are two important differences. The first difference is that, the properties verified during behavior verification are not restricted to invariants. The other difference is that our approach is modular. During interface verification, we only check the correct usage of the concurrency controllers. Since the controller is guaranteed to satisfy the given synchronization properties, after behavior verification, interface verification does not have to search for synchronization errors and does not have to generate all possible interleavings of the concurrent threads.

To avoid the error-prone usage of low-level synchronization primitives, the recently released J2SE 5.0 includes a concurrency utilities package [28]. The package involves a `Lock` interface and a `ReadWriteLock` among other utilities. Similar to our framework, developers can create their own synchronization policies by implementing these interfaces. The verification approach enabled by the concurrency controller pattern can be adapted to automated verification of these custom implementations. With the concurrency utilities package, the lock acquisitions in the programs have to be explicit as well. Interface verification can be used to detect errors such as missing lock operations and unprotected data access.

Model checking finite state abstractions of programs has been studied by several researchers [11, 2, 12, 20, 30]. We present a modular verification approach where behavior and interface checking are separated based on the interface specification provided by the programmer. Also, we use infinite state verification techniques for behavior verification instead of constructing finite state models via abstraction.

Yavuz-Kahveci et al. [42] specify concurrency controllers directly in Action Language. We eliminate the overhead of writing specifications in a specification language by introducing the concurrency controller pattern. Also, the authors do not address controller interfaces and interface verification to check the assumptions of concurrency controller behavior on the code.

This paper builds on our earlier work on the concurrency controller pattern [5] and design for verification [8]. Our main contribution in the current paper is to provide a formal model for behavior and interface verification techniques. This model enables us to formalize the modular verification strategy that is crucial for the scalability of the presented approach.

7. Conclusion

Our results in this paper demonstrate that model checking can be effective in verification of synchronization operations in concurrent programs when it is combined with a design for verification approach that facilitates scalable verification. The design for verification approach presented in this paper builds on the concurrency controller pattern. Concurrency controller pattern enables modular verification of concurrency controllers by decoupling their behaviors and interfaces. Modularization of the verification task improves its efficiency and enables us to combine different verification techniques with their associated strengths. We believe that the presented design for verification approach can be extended to other application domains using behavioral design patterns that support stateful interfaces and by developing modular verification techniques that exploit such interfaces.

References

[1] R. Alur, Pavol Cerny, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *Proceedings of the 32nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 98–109, 2005.

[2] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of SPIN Workshop on Model Checking Software*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122, 2001.

[3] Thomas Ball and Sriram K. Rajamani. The slam project: Debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 1–3, Portland, USA, January 2002.

[4] A. Betin-Can and T. Bultan. Interface-based specification and verification of concurrency controllers. Technical Report 2003-13, Computer Science Department, University of California, Santa Barbara, June 2003.

[5] A. Betin-Can and T. Bultan. Verifiable concurrent programming using concurrency controllers. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE)*, pages 248–257, 2004.

[6] 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 *Proceedings of the 20th IEEE International Conference on Automated Software Engineering (ASE)*, pages 14–23, 2005.

[7] Jeffery George Bogda. *Program Analysis Alleviates Java Synchronization*. PhD thesis, University of California, Santa Barbara, 2001.

[8] T. Bultan and A. Betin-Can. Scalable software model checking using design for verification. In *Proceedings of the IFIP Working Conference on Verifi ed Software: Theories, Tools, Experiments*, 2005.

[9] T. Bultan and T. Yavuz-Kahveci. Action language verifier. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, pages 382–386, 2001.

[10] T. Cargill. Specific notification for Java thread synchronization. In *Proceedings of the 3rd Conference on Pattern Languages of Programs*, 1996.

[11] J. C.Corbett, M. B.Dwyer, J. Hatcliff, S. Laubach, C. S. Pasarenau, Robby, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*, pages 439–448, 2000.

[12] Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. In *Proceedings of the 25th International Conference on Software Engineering*, pages 385–395, 2003.

[13] A. Chakrabarti, L. de Alfaro, T.A. Henzinger, M. Jurdziński, and F.Y.C. Mang. Interface compatibility checking for software modules. In *Proceedings of the 14th International Conference on Computer Aided Verifi cation*, Lecture Notes in Computer Science 2404, pages 428–441. Springer-Verlag, 2002.

[14] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for java. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, volume 34, October 1999.

[15] E.M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[16] R. DeLine and M. Fahndrich. Enforcing high-level protocols in low-level software. In *Proc. 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 59–69, 2001.

[17] R. DeLine and M. Fahndrich. Typestates for objects. In *Proceedings of the 18th European Conference on Object-Oriented Programming ECOOP*, pages 465–490, 2004.

[18] G. Delzanno. Automatic verification of parameterized cache coherence protocols. In *Proc. 12th International Conference on Computer Aided Verifi cation*, volume 1855 of *Lecture Notes in Computer Science*, pages 53–68, 2000.

[19] X. Deng, M. B. Dwyer, J. Hatcliff, and M. Mizuno. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *Proceedings of the 24th International Conference on Software Engineering*, pages 442–452, 2002.

[20] Matthew B. Dwyer, John Hatcliff, Roby Joehanes, Shawn Laubach, Corina S. Pasareanu, Robby, Willem Visser, and Hongjun Zheng. Tool-supported program abstraction for finite-state verification. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 177–187, 2001.

[21] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Proceedings of the 29th ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 234–245, 2002.

[22] Cormac Flanagan and Shaz Qadeer. Thread-modular model checking. In *Proceedings of the Tenth International Workshop on Model Checking of Software (SPIN)*, pages 213–224, 2003.

[23] P. Godefroid, C. Colby, and L. Jagadeesan. Automatically closing open reactive programs. In *Proceedings of 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 345–357, June 1998.

[24] K. Havelund, M. Lowry, and J. Penix. Formal analysis of a space craft controller using spin. *IEEE Transactions on Software Engineering*, 27(8):749–765, August 2001.

[25] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Software verification with blast. In *Proceedings of the Tenth International Workshop on Model Checking of Software (SPIN)*, Lecture Notes in Computer Science 2648, pages 235–239. Springer-Verlag, 2003.

[26] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[27] Indus. http://indus.projects.cis.ksu.edu.

[28] Java 5.0 concurrency utilities. http://java.sun.com/j2se/1.5.0/docs/guide/concurrency/.

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

[30] J. Magee and J. Kramer. *Concurrency: State Model and Java Programs*. Wiley, 1999.

[31] P.C. Mehlitz. Design for verification with dynamic assertions. Technical report, NASA Ames, July 2003.

[32] P.C. Mehlitz and J. Penix. Design for verification using design patterns to build reliable systems. In *Proceedings of 6th Workshop on Component-Based Software Engineering*, 2003.

[33] O'Reilly. *Java Distributed Computing*. O'Reilly and Associates Inc., Sebastopol, California, 1998.

[34] C. S. Pasareanu, M. B. Dwyer, and M. Huth. Assume guarantee model checking of software: A comparative case study. In *Theoretical and Practical Aspects of SPIN Model Checking*, volume 1680 of *Lecture Notes in Computer Science*, pages 168–183, 1999.

[35] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. Wiley and Sons, 2000.

[36] N. Sharygina, J. C. Browne, and R. P. Kurshan. A formal object-oriented analysis for software reliability: Design for verification. In *Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 318–332, 2001.

[37] Oksana Tkachuk and Matthew B. Dwyer. Adapting side-effects analysis for modular program model checking. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE)*, pages 116–129, 2003.

[38] Oksana Tkachuk, Matthew B. Dwyer, and Corina Pasareanu. Automated environment generation for software model checking. In *In the Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference, ESEC/FSE 2003*, pages 188–197, 2003.

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

[40] J. Whaley, M. Martin, and M. Lam. Automatic extraction of object-oriented component interfaces. In *Proceedings of the 2002 ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002)*, pages 218–228, 2002.

[41] T. Yavuz-Kahveci, C. Bartzis, and T. Bultan. Action language verifier, extended. In *Proceedings of the 17th International Conference on Computer Aided Verifi cation (CAV 2005)*, pages 413–417, 2005.

[42] T. Yavuz-Kahveci and T. Bultan. Specification, verification, and synthesis of concurrency control components. In *Proceedings of the 2002 ACM/SIGSOFT International Symposium on Software Testing and Analysis*, pages 169–179, 2002.