Verifiable Concurrent Programming Using Concurrency Controllers

Aysu Betin-Can and Tevfik Bultan
Computer Science Department
University of California, Santa Barbara, CA 93106, USA
{aysu,bultan}@cs.ucsb.edu

Abstract

We present a framework for verifiable concurrent programming in Java based on a design pattern for concurrency controllers. Using this pattern, a programmer can write concurrency controller classes defining a synchronization policy by specifying a set of guarded commands and without using any of the error-prone synchronization primitives of Java. We present a modular verification approach that exploits the modularity of the proposed pattern, i.e., decoupling of the controller behavior from the threads that use the controller. To verify the controller behavior (behavior verification) we use symbolic and infinite state model checking techniques, which enable verification of controllers with parameterized constants, unbounded variables and arbitrary number of user threads. To verify that the threads use a controller in the specified manner (interface verification) we use explicit state model checking techniques, which allow verification of arbitrary thread implementations without any restrictions. We show that the correctness of the user threads can be verified using the concurrency controller interfaces as stubs, which improves the efficiency of the interface verification significantly. We also show that the concurrency controllers can be automatically optimized using the specific notification pattern. We demonstrate the effectiveness of our approach on a Concurrent Editor implementation which consists of 2800 lines of Java code with remote procedure calls and complex synchronization constraints.

1. Introduction

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. [13].

In this paper, we propose using concurrency controller classes to coordinate the interaction among multiple threads. A concurrency controller class encapsulates the variables and actions required for concurrency control and also serves as a specification of the synchronization policy. We define a behavioral design pattern called *concurrency controller pattern* to facilitate developing concurrency controllers. We show that both safety and liveness properties of concurrency controllers can be automatically verified using a modular verification approach which decouples the verification of the controller properties (behavior verification) from the verification of the threads which use the controller (interface verification). 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.

To implement a synchronization policy based on the presented design pattern, software developer needs to write a set of controller *actions* where each action is specified as a set of guarded commands. The presented approach also requires the developer to write a *controller interface* which specifies the order in which the actions of the controller can be executed by the threads which use the controller. The controller interface corresponds to a finite state machine with transitions representing controller actions. We verify the behavior of the concurrency controller by assuming that the threads which use the controller obey the controller interface. During interface verification we verify this assumption.

For behavior verification we use an infinite state symbolic model checker called the Action Language Verifier [4] by *automatically* generating Action Language specifications from the concurrency controller classes. For interface verification we use the explicit and finite state model checker Java PathFinder (JPF) [3] and verify that each thread that use the concurrency controller classes obey the corresponding controller interface. Note that, controller interfaces have states and cannot be specified as Java interfaces. In the concurrency controller pattern, we use Java classes to represent controller interfaces. These interface classes serve as stubs during interface verification, significantly improving the performance.

Related Work: In [8] interfaces of software modules are specified as a set of constraints, and algorithms for interface compatibility checking are presented. Our approach to interface checking can be seen as a special case of the interface compatibility checking. However, we do not require each method to be annotated with interface constraints as in [8]. Also, unlike [8], our goal is to verify both the interfaces and the behaviors of the concurrency controllers.

In [10], Deng et al. present an approach for synthesizing synchronization code in concurrent programs from invariant specifications. 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 which serve as stubs. Although these approaches may seem similar, there are two important differences: 1) During behavior verification, we can handle all ACTL properties including liveness properties, not just invariants. 2) 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 generate all possible interleavings of the concurrent threads.

Model checking finite state abstractions of programs has been studied in [6, 1, 7]. 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.

In [14] monitors are modeled with guarded actions and verified using finite state verification techniques. Java implementation of these models is also discussed. Finite state verification techniques used in [14] cannot handle models with parameterized constants and unbounded variables and, most importantly, for arbitrary number of threads. Moreover, in [14] correct usage of the monitors by the threads is not addressed.

Design patterns for multi-threaded systems have been studied in [18, 11, 13, 19]. These patterns are developed to help concurrent programmers in writing reliable concurrent programs. Our goal, however, is to present a design pattern which improves the verifiability of concurrent programs by automated tools. In addition to presenting a verifiable design pattern for concurrency, we also present a modular verification technique that exploits the presented pattern.

In [15] the authors suggest the design for verification approach which promotes using design patterns and exploiting the properties of the used patterns in improving the efficiency of the automated verification. Our approach in this paper can also be interpreted as an instance of design for verification approach for concurrent programming.

Our work in this paper builds on the approach presented in [20] and [2]. In [20], concurrency controllers are specified directly in Action Language and controller interfaces and interface verification are not addressed. In [2], behaviors and interfaces of concurrency controllers are specified in a simple guarded-command language. In the approach presented in this paper, we eliminate the overhead of writing specifications in a specification language by introducing the concurrency controller pattern. We also present a case study to show the effectiveness of the concurrency controller pattern.

The rest of the paper is organized as follows: Section 2 describes the Concurrent Editor application and explains the design forces for the concurrency control pattern, Section 3 presents the concurrency controller pattern, Section 4 discusses behavior and interface verification, Section 5 discusses the implementation and the verification of the Concurrent Editor, and Section 6 presents our conclusions.

2. Case Study: Concurrent Editor

To demonstrate the effectiveness of our approach on realistic concurrent programs, we conducted a case study. The Concurrent Editor is a complex application that requires remote procedure calls and concurrent accesses to shared data. In this section, we give a description of this application, the difficulties that can arise during the implementation of such a system, and the design forces leading to the concurrency controller pattern.

Requirements: The Concurrent Editor is 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 (Figure 1).

The Concurrent Editor allows multiple users to edit a document concurrently as long as they are editing different paragraphs and 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) (Figure 2 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 changes indicating 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

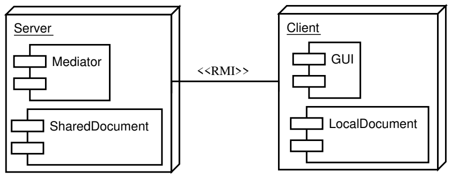

Figure 1: Concurrent Editor Architecture

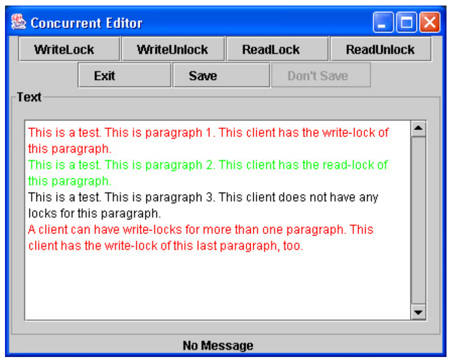

Figure 2: Concurrent Editor Screen Shot

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.

Design Issues: 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 collaborative infrastructure where all client nodes register to a mediator and messages are broadcasted to the client nodes through the mediator [16] (see Figure 1).

To handle the concurrent access to the shared document a number of synchronization issues need to be considered. 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 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 effecting the synchronization policy. Similarly, one synchronization policy could be replaced with another without changing the paragraph implementation. We can summarize these observations by the following design forces: *1) The synchronization policies should be pluggable*; *2) Shared data classes should be maintainable.*

An efficient implementation of a synchronization policy should not awaken all the waiting threads after every operation. For example, a thread waiting for read access should only be awakened after a write-unlock operation. This requirement can be summarized as another design force: *3) There should be a mechanism to prevent unnecessary context-switch among threads.* However, efficient implementation of synchronization policies in Java requires conditional waits and notifications implemented with multiple locks and multiple condition variables which is error-prone. This leads to the next design force: *4) There should be a high-level synchronization construct for implementing synchronization policies which does not sacrifice efficiency.*

Above design forces can be observed in other synchronization problems that arise in implementing the Concurrent Editor. We will describe another one here. 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 mutex lock is not enough. We need a conditional wait for worker threads when the buffer is empty. We also need to bound 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. Note that the design forces we discussed above are also valid for the implementation of this synchronization policy.

Finally, even if all the above design forces are resolved by a design pattern and high level synchronization constructs are used to implement synchronization policies, due to complex synchronization constraints, an implementation of Concurrent Editor is still prone to errors. There should be a scalable automated verification framework which ensures that the synchronization constraints are met. This is summarized as our last design force for the concurrency controller pattern: *5) The implementation should be verifiable.*

The concurrency controller pattern presented in this paper resolves the above design forces. In this pattern synchronization policies are implemented using guarded commands preventing error-prone usage of synchronization statements. The presented pattern separates the synchronization operations from the operations that change the shared object's

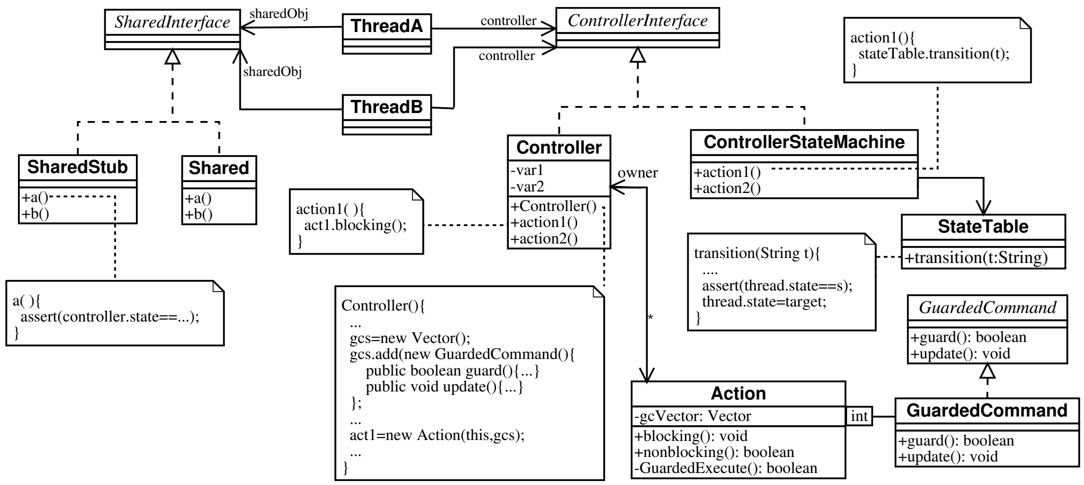

Figure 3: Concurrency Controller Pattern Class Diagram

internal state. This decoupling makes the synchronization policy pluggable and improves the maintainability of the code. We exploit this decoupling in our modular verification technique based on controller interfaces. This 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 [5].

3. Concurrency Controller Pattern

Figure 3 shows the class diagram for the concurrency controller pattern. The `ControllerInterface` is a Java interface which defines the names of actions available to the user 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 which specifies the order in which the actions of the controller can be executed by the threads.

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 Figure 3, the programmer 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 of the corresponding action. A blocking action causes the calling thread to wait until one of the guarding conditions become true.

Consider the reader-writer synchronization (`RW`) used in

```
class RWController implements RWInterface{
 int nR; boolean busy;
 final Action act_r_enter, act_r_exit;
 final Action act_w_enter, act_w_exit;
 RWController() {
  ...
  gcs = new Vector();
  gcs.add(new GuardedCommand() {
   public boolean guard(){
     return (nR == 0 && !busy );}
   public void update(){
     busy = true; } });
  act_w_enter = new Action(this,gcs);

  gcs = new Vector();
  gcs.add(new GuardedCommand() {
   public boolean guard(){ return true;}
   public void update(){ busy = false; } });
  act_w_exit= new Action(this,gcs);
}
public void w_enter(){ act_w_enter.blocking();}
public boolean w_exit(){return act_w_exit.nonblocking();}
 ...
}
```

Figure 4: An excerpt from RWController class

the Concurrent Editor. A controller for `RW` can be implemented using four actions and two variables. The variables are `nR`, denoting the number of readers in the critical section, and `busy`, denoting if there is a writer in the critical section. The `RWController` in Figure 4 is the controller class for `RW`.

Controller Behavior: We specify the behavior of a concurrency controller in a guarded command style similar to that of CSP [12]. Since Java language does not have a guarded command structure we provide the `GuardedCommand` interface and the `Action` class. Each instance of the `Action` class has a vector of guarded commands which defines its behavior (see Figure 3).

The code for the `Action` class is given in Figure 5. 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, this method returns `false`. The execution of a blocking action is implemented by the `blocking` method. When a thread calls a blocking ac-

```
public class Action{
 protected final Object owner;
 private final Vector gcV;
 public Action(Object c, Vector gcs){...}
 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(); }
 }
}
```

Figure 5: Action Class

tion, it has to execute a guarded command. Therefore, if the `GuardedExecute` method does not execute one of the guarded commands, 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. A nonblocking action does not cause the calling thread to wait. A nonblocking action returns `true` and notifies the other threads if a guarded command is successfully executed. It returns `false` if the guards of all its guarded commands are false.

Figure 6 shows a sequence diagram demonstrating the use of the concurrency controller pattern. In this scenario, thread `B` calls the controller action `action1`, which is a blocking action. After thread `B` executes the blocking action successfully, thread `A` calls the same action, however, thread `A` is blocked by the controller. While thread `A` is blocked, thread `B` successfully executes a couple of operations on the shared object. After it finishes its operations on the shared object, thread `B` calls `action2` (a nonblocking action of the controller). The last controller action executed by thread `B` enables the action that is blocking thread `A`, and thread `A` successfully completes executing `action1`. For example, this sequence of events corresponds to two threads interacting using a mutex lock where `action1` corresponds to *acquire* action and `action2` corresponds to *release* action.

Controller Interface: The interface of a concurrency controller defines the acceptable call sequences for the threads that use the controller. These allowed call sequences are specified using a finite state machine which is shown as the `ControllerStateMachine` class in Figure 3.

The interfaces of the concurrency controllers used in the Concurrent Editor is shown in Figure 7 (a), (b), and (c). Consider the state machine shown in Figure 7(a) representing the interface of the `RW` controller. In our framework, a user can specify this controller interface as `RWStateMachine` shown in Figure 8. The state machine has three states `idle`, `reading`, and `writing` encoded as integer constants. The transitions are defined in the constructor and the methods of the controller interface invokes these transitions, e.g., the `w_enter` method invokes the transition from `idle` to `writing`.

The attribute `stateTable` holds the current interface state of each thread that uses the controller. The implementation for `StateTable` is the same for each controller interface. The `transition(action)` method in this class asserts that the current thread is in one of the legal states from which there exists a transition on `action`, and sets the the thread's state to the target state of that transition.

The interfaces of concurrency controllers can be complex. For example, the state machine in Figure 7(d) is the interface of a concurrency controller for an Airport Ground Traffic Control simulation program [20]. This controller consists of 13 integer variables and 20 actions. The controller actions are called to simulate the behavior of an airplane in an airport ground network model similar to that of the Seattle/Tacoma International Airport.

Controller Semantics: A concurrency controller can be defined as a tuple $CC = (V, IC, A, F)$, where V is the set of variables, IC is the initial condition, A is the set of actions, and F is the controller interface. For example, for the RW controller discussed above, $V = \{\mathtt{nR}, \mathtt{busy}\}$ and $A = \{\mathtt{r\_enter}, \mathtt{r\_exit}, \mathtt{w\_enter}, \mathtt{w\_exit}\}$. The initial condition denotes the initial values assigned to the variables in the constructor of the controller class. Formally, IC is a predicate on the variables in V, i.e., $IC : \prod_{v \in V} \mathrm{DOM}(v) \to \{\mathrm{TRUE}, \mathrm{FALSE}\}$, where $\mathrm{DOM}(v)$ denotes the domain of the variable v and $\prod_{v \in V} \mathrm{DOM}(v)$ denotes the Cartesian product of the variable domains. For the RW $IC \equiv \mathtt{nR} = 0 \wedge \neg\mathtt{busy}$. The interface of a concurrency controller is a finite state machine $F = (IF, SF, RF)$ where SF is the set of states of the interface, $IF \in SF$ is the initial state of the interface and $RF \subseteq SF \times A \times SF$ is the transition relation of the interface. Each transition in the interface is labeled with an action of the controller.

The semantics of a concurrency controller specification

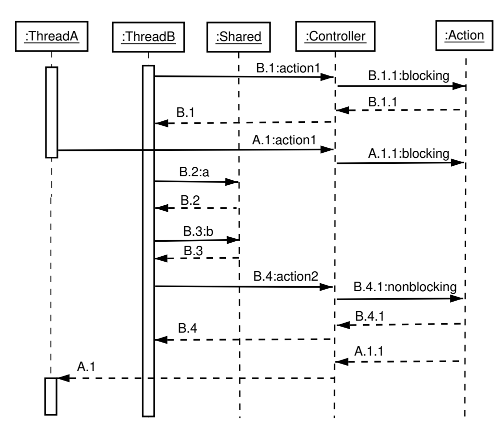

Figure 6: Controller Sequence Diagram

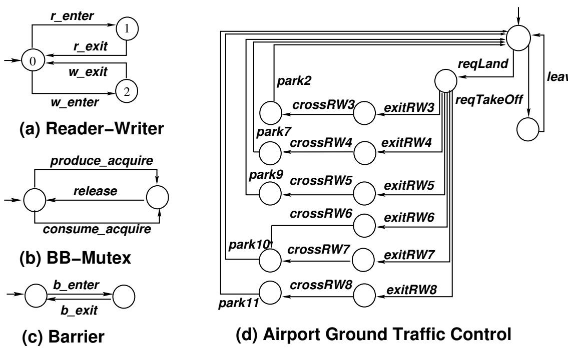

Figure 7: Controller Interfaces

```
public class RWStateMachine implements RWInterface{
 StateTable stateTable;
 final static int idle=0, reading=1, writing=2;
 public RWStateMachine(){ ...
   stateTable.insert("w_enter",idle,writing);
 }
 public void w_enter(){
   stateTable.transition("w_enter");
 }
 ...
}
```

Figure 8: Controller Interface for RW

CC is a transition system $T(CC)(n) = (IT, ST, RT)$ where n is the parameter denoting the number of user 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.

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

We introduce the following notation. Given a state $s \in ST$, $s(V) \in \prod_{v \in V} \text{DOM}(v)$ denotes the valuations of the variables in the state s. Given a state s and a thread t, where $1 \leq t \leq n$, $s(SF)(t) \in SF$ denotes the state of the thread t in s, and $s(SF - t) \in \prod^{n-1} SF$ denotes the projection of the state s to the states of all the threads except t.

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

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

Consider a transition $(q, a, q') \in RF$ where q and q' are two interface states, and a is an action. We will define a transition relation $RT_{(q,a,q')} \subseteq ST \times ST$, which corresponds to executing the action a at interface state q. We define $RT^{u}_{(q,a,q')}$, the transition relation for a transition (q, a, q') when one guarded command is executed, as

$$RT^{u}_{(q,a,q')} = \{(s, s') \mid s, s' \in ST \wedge (\exists 1 \leq t \leq n, s(SF)(t) = q \wedge s'(SF)(t) = q' \ \wedge \ s'(SF - t) = s(SF - t)) \wedge (\exists gc \in a.GC, gc.g(s(V)) \ \wedge \ s'(V) = gc.u(s(V)))\}$$

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

$$RT^{nb}_{(q,a,q')} = \{(s, s') \mid s, s' \in ST \wedge \ s'(V) = s(V) \wedge (\exists 1 \leq t \leq n, s(SF)(t) = q \wedge \ s'(SF)(t) = q' \wedge s'(SF - t) = s(SF - t)) \ \wedge \ (\forall gc \in a.GC, \ \neg gc.g(s(V)))\}$$

The transition relation RT of the transition system $T(CC)(n)$ is defined as $RT = \bigcup_{(q,a,q') \in RF} RT_{(q,a,q')}$.

We define the 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 $\forall i \geq 0,\ (s_i, s_{i+1}) \in RT$. Let AP denote the set of atomic properties, where a property $p \in AP$ is a predicate on variables in V, $p : \prod_{v \in V} \text{DOM}(v) \rightarrow \{\text{TRUE}, \text{FALSE}\}$. We use ACTL to state properties of the transition system $T(CC)(n)$. A concurrency controller CC satisfies an ACTL 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.

Optimizing Concurrency Controllers: Concurrency controllers written based on the design pattern given in Figure 3 may be inefficient because of the following reasons: 1) The pattern in Figure 3 does not use the specific notification, hence, after every state change in the controller all the waiting threads are awakened, increasing the context-switch overhead; 2) The inner classes used in the pattern in Figure 3 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 [5], 2) does not have any inner classes, and 3) minimizes the number of method invocations.

Implementing the specific notification pattern requires a notification dependency analysis which can be difficult and complicated to do manually. In our automated optimization process we use the algorithm presented in [20] to compute these dependencies automatically. Below is an excerpt from the optimized version of RW controller generated from the source given in Figure 4.

```
class RWController implements RWInterface{
 ...
 private boolean Guarded_w_enter(){
```

```
  boolean result=false;
  synchronized(this) {
   if(nR==0 && !busy){ busy=true; result=true;} else; }
  return result;
 }
 public void w_enter(){
  synchronized(Condw_enter){
   while (!Guarded_w_enter()){
    try{ Condw_enter.wait();
    } catch(InterruptedException e){;}}}
 }
 public boolean w_exit(){ ...
  //notifies Condr_enter, Condw_enter
 }
}
```

4. Verification of Concurrency Controllers

In this Section, we will present our modular verification approach for concurrency controllers which consists of two phases: behavior verification and interface verification. During behavior verification we verify the controller properties assuming that the user threads adhere to the controller interface. During interface verification we check that each thread which uses a controller obeys the interface of that controller.

Behavior Verification: We verify the behavior of a concurrency controller using the Action Language Verifier [4]. We automatically translate the concurrency controllers (written based on the concurrency controller pattern shown in Figure 3) into Action Language.

An atomic action in the Action Language defines one execution step using current-state values for the variables (denoted as unprimed variables) and next-state values (denoted as primed variables). Our tool translates the guard expression 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 `w_enter` action in the `RWController` class given in Figure 4. In the constructor, this action is defined with one guarded command. The guard expression of this guarded command is `(nR==0 && !busy)`, and the update expression is `busy=true`. In the interface of the controller (Figure 8) the transition associated with `w_enter` changes the thread's state from `idle` to `writing`. Using this information, the translator constructs the atomic action `w_enter` shown in the automatically generated Action Language specification below:

```
module main()
 integer nR; boolean busy;
 module RW()
  enumerated pc {idle,reading,writing};
  initial: nR=0 and busy=true and pc=idle;
  w_enter: pc=idle and nR=0 and !busy
        and busy'=true and pc'=writing;
  ...
  RW: r_enter | r_exit | w_enter | w_exit;
 endmodule
 main: RW()| RW();
 spec: AG([busy => nR=0]) // CTL property
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 shared variables of the controller. In the above example, the `main` module has a submodule called `RW`. Submodule `RW` models the user threads and corresponds to a process type (a thread class in Java). Submodule `RW` has one local enumerated variable (`pc`) which keeps track of the thread state. A thread can be in one of the interface states. Each instantiation of a module will create different instantiations of its local variables.

The ACTL properties that we expect the system to satisfy are written using the `spec` keyword at the end of the `main` module. There are two ways to specify the ACTL properties. One way is writing these properties directly in the generated Action Language specification. The other way is specifying the property in the controller class as annotation via `//@property Expr` comment, where `Expr` is an ACTL formula with predicates written as Java boolean expressions on controller variables, e.g., `AG(!busy || nR==0)`.

Since Action Language Verifier can handle infinite state systems we are able to verify controllers with unbounded integer variables (such as the reader-writer controller) or parameterized constants (such as the bounded-buffer controller). However, currently, Action Language only supports integer, boolean and enumerated types. In order to translate the concurrency controllers written in Java (based on the concurrency controller pattern) to Action Language, we restrict the controller variables to these types (for enumerated variables we use static integers in Java.) Since variables of the concurrency controllers only need to store the state information required for concurrency control, these basic types are sufficient for modeling a wide range of concurrency controllers (including all the controllers mentioned in this paper). The presented approach can be extended to other data types by either extending the Action Language Verifier by adding a symbolic representation for that data type, or by restricting the domain of the new data type to finite domains and using a boolean encoding.

We use an automated abstraction technique, called counting abstraction [9], to verify the behavior of a concurrency controller for arbitrary number of threads. Implementation of counting abstraction for Action Language specifications is discussed in [20]. The basic idea is to define an abstract transition system in which the local states of the threads (corresponding to the states of the 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. (The names of the new variables are obtained by prefixing the name of the state they represent with $c\_$, e.g., c_idle denotes the variable introduced for the interface state `idle`). The initial states and the transition relation of the parameterized system can be defined using linear arithmetic constraints on these new variables [20].

Table 1 shows a set of concurrency controllers and several properties we verified using Action Language Verifier. The instances marked with P are verified for arbitrary number of threads using counting abstraction. The controllers are: RW is the reader-writer controller (We have considered two different properties of RW, denoted as RW-1 and RW-

Table 1: Controller Properties

Controller	Property
MUTEXP	$AG(c\_cs \geq 0 \wedge c\_cs \leq 1)$
BB	$AG(count \geq 0 \wedge count \leq size)$
BB-MUTEXP	$AG((count = x \wedge c\_cs > 0) \Rightarrow AX(count = x))$
BARRIERP	$AG((c\_cs = x \wedge count < limit) \Rightarrow AX(c\_cs \geq x))$
RW-1	$AG(busy \Rightarrow nR = 0)$
RW-2	$AG(busy \Rightarrow AF(\neg busy))$
BB-RW	$AG((nR > 0 \wedge count = x) \Rightarrow AX(count = x))$
AIRPORT	$AG(numRW16R \leq 1 \wedge numRW16L \leq 1)$

Table 2: Controller Verification

Controller Instance	Action Language Verifier		JPF with stubs	
	T1(s)	M1(MB)	T2(s)	M2(MB)
MUTEXP	0.01	6.00	1.69	0.40
BB	0.03	0.61	1.87	1.41
BB-MUTEXP	2.05	6.47	1.88	1.71
BARRIERP	0.01	0.50	1.77	1.05
RW-1	0.01	6.30	1.93	1.37
RW-2	0.01	6.10	1.93	1.37
RWP-1	0.42	6.94	1.93	1.37
RWP-2	0.06	6.16	1.93	1.37
BB-RW	0.13	6.76	2.74	1.43
BB-RWP	0.63	10.80	2.74	1.43
AIRPORT	0.35	23.09	3.33	2.15
AIRPORTP	0.61	31.29	3.33	2.15

2), AIRPORT is the concurrency controller for an Airport Ground Traffic Control simulation program mentioned in Section 3, BB is a bounded buffer, BB-MUTEX is a bounded buffer with a mutex lock, BB-RW is a bounded buffer with a reader-writer lock, and BARRIER is a controller for barrier synchronization. Table 2 shows the performance results for the verification of these concurrency controllers. The columns labeled T1 and M1 denote the time and the memory usage for the behavior verification by the Action Language Verifier.

Interface Verification: User threads of a concurrency controller must adhere to the interface of the concurrency controller. A call sequence $cs = a_0, a_1, \ldots$ is a sequence of action executions by a user thread. A thread is correct with respect to an interface if all the call sequences generated by the thread can also be generated by the finite state machine defining the interface. If all the thread implementations in a concurrent program are correct with respect to a concurrency controller's interface, then the ACTL properties verified on the concurrency controller are preserved by that concurrent program.

We use JPF [3] to check the correctness of the user threads. JPF is an explicit state model checker for Java. It enables us to verify arbitrary Java threads without any restrictions on data types. JPF supports property specifications via assertions that are embedded in the source code. We use the state machines for the controller interfaces as stubs to improve the efficiency of the interface verification. As explained in Section 3, state machines for the controller interfaces use assert statements to check if an action execution is allowed in a given interface state. If a user thread does not obey the interface, JPF reports the associated assertion failure. If no assertion failure is reported, this means that all the call sequences generated by the user threads can also be generated by the controller interface. Since the controller interfaces do not contain variables of the controller, the state space searched by JPF is reduced.

In the interface verification, we also check that a user thread accesses shared data only when it is in a correct interface state. I.e., we want to guarantee that user threads access shared data only through the controller. These rules are implemented in the `SharedStub` class shown in Figure 3. The methods of the `SharedStub` class have assertions on the controller state instead of actual data manipulation operations. The methods of the `SharedStub` class return a randomly chosen value out of all possible return values. We use the random value generator methods of JPF and JPF searches the state space for all possible values generated (i.e., this is an exhaustive search not random testing). We are assuming that the state of the shared data is stored in private fields of the `Shared` class and is accessible only through query methods. In the interface verification, the instances of the shared data classes are replaced by the corresponding `SharedStub` instances.

During interface verification we do not have to worry about thread interactions if the threads interact only through shared variables and controllers. The thread coordination is handled by the controllers and the controllers are verified during behavior verification. Goal of interface verification is to ensure that the order of method calls from a thread to a controller obeys the interface specification of that controller. The states of the shared data may influence the behavior of the threads. However, since we replace the methods of the shared data classes with stubs which force JPF to search the state space exhaustively for every possible return value, during interface verification we conservatively verify that any possible behavior of the thread obeys the interface. Note that the behaviors of the threads are completely decoupled during interface verification. The stubs for the concurrency controllers and the shared data close the environments of the threads. Hence, we do not need to consider all possible thread interleavings during interface verification.

In Table 2, the last two columns (T2 and M2) show the time and memory usage of JPF during the interface verification. The memory usage is low since we use stubs with finite reachable state spaces and we do not have to consider all possible thread interleavings. For the examples reported in Table 2 we verified the thread implementations by running a single instance of each thread class in isolation. JPF successfully verifies the problem instances with stubs without exhausting memory.

To demonstrate the effectiveness of our modular verification approach, we verified some controllers with JPF without using stubs. Table 3 shows two of these examples, BB-RW and AIRPORT. We used both depth-first and breadth-first search heuristics. The column labeled TN-S shows the number of user threads and the buffer size (for BB-RW). The memory usage is shown in the column labeled M, and the execution time is displayed in the column labeled T. For the BB-RW case it is not possible to verify the original specification using a program checker such as JPF since the size

Table 3: JPF Without Stubs

Instance	TN-S	DFS T(s)	DFS M(MB)	BFS T(s)	BFS M(MB)
BB-RW	2 – 1	23.65	23.81	48.67	17.11
BB-RW	2 – 2	63.89	57.53	161.11	55.58
BB-RW	2 – 3	174.35	150.11	505.76	141.28
BB-RW	2 – 4	520.71	333.06	1542.68	329.13
BB-RW	2 – 5	↑	↑	↑	↑
BB-RW	3 – 2	↑	↑	↑	↑
AIRPORT	2	25.79	24.61	57.39	23.59
AIRPORT	3	1430.44	329.71	880.37	309.71
AIRPORT	4	↑	↑	↑	↑

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 3. JPF runs out of (512MB) memory (denoted by ↑) for buffer size 5 and 2 user threads for both heuristics because of the state space explosion. For the AIRPORT case the performance of JPF drops dramatically when the number of user threads increases because of the increase in the number of possible interleavings. JPF runs out of 512MB memory for 4 user threads for this example.

5. Concurrent Editor Revisited

In this section we present an implementation of the Concurrent Editor we discussed in Section 2 using concurrency controllers and discuss how we verified the implementation using the presented verification approach. Figure 9 shows the class diagram of the system. 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 coordinating the read and write accesses to that paragraph. This node also has a barrier controller which is used whenever a user wants to save the document.

A client node has a GUI with an editable text area and seven buttons (see Figure 2). The text area is of type `JTextPane` with 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. The event thread is automatically created by the Java Virtual Machine (JVM) to capture the events related to the GUI [17]. 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 handles the communication with 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 message buffer associated with that paragraph. The access to this buffer is controlled by a bounded-buffer mutex controller.

The Concurrent Editor implementation consists of 2800 lines of Java code with 17 class files, 5 controller classes, and 7 Java interfaces. We used concurrency controllers to coordinate the interaction among multiple threads. Hence,

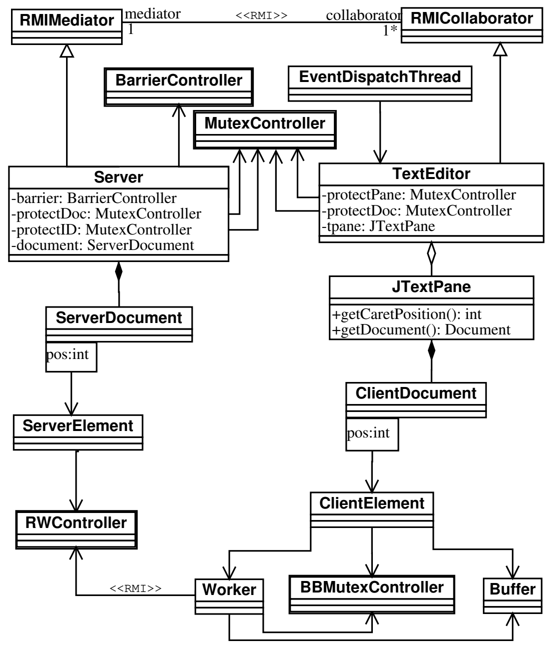

Figure 9: Concurrent Editor Class Diagram

we implemented the Concurrent Editor without writing any Java synchronization operations. The behaviors of the controllers we used in the Concurrent Editor implementation have been verified using the Action Language Verifier (behavior verification results are given in Section 4).

We performed interface verification on the Concurrent Editor to ensure the correct usage of the concurrency controllers. However, JPF can not handle the native RMI methods used in the Concurrent Editor implementation. Note that an application with remote procedure calls is a collection of programs running on different JVMs. We can isolate one node by replacing the remote nodes with a local driver simulating the incoming calls from remote nodes, and with local stubs simulating the outgoing calls to remote nodes. For example, Concurrent Editor is a collection of two programs, i.e., the server and the client. To prepare the server node for interface verification, we created a driver class which calls every remotely accessible method of the server. We also created a stub which has the same remote method signatures as the client node. Whenever the server program needs to invoke a remote method of the client program, it invokes the corresponding stub method. When a stub method is invoked, it either throws a `RemoteException` or returns a randomly chosen possible value (using random value generator methods in JPF and JPF exhaustively checks all possible values generated by such methods). As described in Section 4, the instances of the controllers are also replaced with the corresponding stubs (i.e., the interface state machines), and the shared data protected by these controllers are replaced with the corresponding data stubs. Some of these shared data classes were `Document` and `Element` classes in the `javax.swing.text` package and JPF cannot handle these classes due to their native codes. The use of data stubs avoids this problem. The interface verification of the server node took 185.85 seconds and used 67.14 MB memory.

For the interface verification of the client program we implemented a driver class to simulate the behavior of the server node. Client side also interacts with the user through GUI events which are captured by the event thread. We implemented a driver that generates a sequence of user events and simulates the execution of the event thread for each event sequence. In verifying the event thread, we faced another difficulty because the worker threads are created by the event thread through calls made to the document classes. Therefore, the stubs we implemented for these classes both check the controller interface state and perform the actual data operations (we eliminated the native calls in these classes). Other than the creation of the worker threads, the interaction between the event thread and the worker threads are only through shared data and controllers. Therefore the controller interfaces and the data stubs are sufficient to close the environment of the worker threads. The interface verification for the client node implementation with respect to the server driver took 229.18 seconds and used 127.04 MB memory. The interface verification of the event thread took 1636.62 seconds and used 139.48 MB memory. The interface verification of the worker thread took 19.47 seconds and used 5.36 MB memory.

In the interface verification phase, 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 with the Concurrent Editor suggests that the approach we present in this paper is scalable to large, realistic systems.

6. Conclusions

We introduced a concurrency controller pattern for verifiable concurrent programming in Java. Based on this pattern, we presented a modular approach to verification of concurrency controllers by decoupling their behaviors and interfaces. Modularization of the verification task improves the efficiency of the verification and helps us combine different approaches to verification with their associated strengths. We showed that properties of concurrency controllers can be verified with Action Language Verifier by an automated translation of the controller classes to the Action Language. We showed that JPF can be used to verify that the user threads of a concurrency controller adhere to the controller interface. We showed that using controller interfaces as stubs improves the efficiency of the interface verification significantly. We used the proposed framework in the implementation and verification of a Concurrent Editor, demonstrating its scalability.

References

[1] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proc. of SPIN Workshop on Model Checking Software*, 2001.

[2] A. Betin-Can and T. Bultan. Interface-based specification and verification of concurrency controllers. In *Proc. of the Workshop on Software Model Checking, Electronic Notes in Theoretical Computer Science (ENTCS)*, volume 89, 2003.

[3] G. Brat, K. Havelund, S. Park, and W. Visser. Java pathfinder: Second generation of a Java model checker. In *Proc. Workshop on Advances in Verification*, 2000.

[4] T. Bultan and T. Yavuz-Kahveci. Action language verifier. In *Proc. 16th IEEE Int. Conf. on Automated Soft. Eng.*, 2001.

[5] T. Cargill. Specific notification for Java thread synchronization. In *Proc. 3rd Conf. on Pattern Lang. of Programs*, 1996.

[6] 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 *Proc. 22nd Int. Conf. on Soft. Eng.*, 2000.

[7] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *Proc. of the 25th Int. Conf. on Soft. Eng. (ICSE 2003)*.

[8] A. Chakrabarti, L. de Alfaro, T. Henzinger, M. Jurdziński, and F. Mang. Interface compatibility checking for software modules. In *Proc. of the 14th Int. Conf. on Computer Aided Verification (CAV 2002)*.

[9] G. Delzanno. Automatic verification of parameterized cache coherence protocols. In *Proc. 12th Int. Conf. on Computer Aided Verification*, volume 1855 of *LNCS*, 2000.

[10] X. Deng, M. B. Dwyer, J. Hatcliff, and M. Mizuno. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *Proc. 24th Int. Conf. on Soft. Eng.*, 2002.

[11] M. Grand. *Patterns in Java Volume 1: A Catalog of Reusable Design Patterns Illustrated with UML, 2nd edition*. Wiley & Sons, 2002.

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

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

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

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

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

[17] O'Reilly. *Java Swing*. O'Reilly and Associates Inc., Sebastopol,California, 1998.

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

[19] A. Silva, J. Pereira, and J. Marques. Object synchronizer: A design pattern for object synchronization. In *Proc. of Europian Conference of Pattern Languages*, 1996.

[20] T. Yavuz-Kahveci and T. Bultan. Specification, verification, and synthesis of concurrency control components. In *Proc. 2002 ACM/SIGSOFT Int. Symp. on Soft. Testing and Analysis*, pages 169–179, 2002.