Interface-Based Specification and Verification of Concurrency Controllers *

Aysu Betin-Can and Tevfik Bultan
Computer Science Department
University of California
Santa Barbara, CA 93101, USA
{aysu,bultan}@cs.ucsb.edu

Abstract

We present a modular approach to specification and verification of concurrency controllers by decoupling their behavior and interface specifications. The behavior specification of a concurrency controller defines how its shared variables change their values whereas the interface specification defines the order in which a client thread should call its methods. We show that the concurrency controllers can be designed modularly by composing their interfaces. We separate the verification of the concurrency controllers from the verification of the threads that use them. For the verification of the concurrency controllers we use infinite state verification techniques which enable us to verify controllers with parameterized constants and arbitrary number of user threads. We automatically generate Java monitors from the concurrency controller specifications which preserve the verified properties. For the thread verification we use finite state program verification tools which enable us to verify Java threads without any restrictions. We show that the user threads can be verified using stubs generated from the concurrency controller interfaces which improves the efficiency of the thread verification significantly.

1 Introduction

Run-time errors in concurrent programs are generally due to wrong usage of synchronization primitives such as monitors. Conventional validation techniques such as testing become ineffective for concurrent programs since the state space increases exponentially with the number of concurrent threads. Recently, concurrent programming using monitors gained increased attention [Lea99] since monitors are an integral part of Java. In Java each object has a lock associated with it which can be used for synchronization among multiple execution threads. Threads can implement conditional waits using `wait` and `notify` methods. Coordinating wait and notify operations and multiple locks among multiple threads can be very challenging.

In this paper, we propose a modular approach to specification and verification of concurrency controllers by decoupling their behaviors and interfaces. A *concurrency controller* is a synchronization primitive for coordinating concurrent execution among multiple threads via shared variables. Within the scope of this paper we allow shared variables of the concurrency controllers to have the following types: boolean, enumerated, and integer. Behavior specification of a concurrency controller defines how its shared variables change their values. We specify the behavior of a concurrency controller as a set of actions where each action consists of a set of guarded commands. These actions are called by concurrent threads to coordinate their execution. We do not require the specification of condition variables and the associated notify and wait statements as in monitors. These are generated automatically from the concurrency controller specification. An interface specification of a concurrency controller defines the order in which a client thread should call its actions. We specify the concurrency controller interfaces as finite state machines where each transition represents an

*This work is supported in part by NSF grant CCR-9970976 and NSF CAREER award CCR-9984822.

action execution. Different concurrency controllers can be composed to form more complex concurrency controllers by composing their interfaces. We show that if the composed interface is a refinement of the original interface, then the ACTL properties of the original concurrency controller is preserved by the composition.

We use both symbolic and explicit state automated verification techniques by exploiting the separation between the behavior and the interface of a concurrency controller. We verify the behavior of a concurrency controller independent of the threads that use it based on its behavior and interface specifications using symbolic and infinite state verification techniques. The interface specification provides the most general context that the behavior of the concurrency controller has to be verified against. The responsibility of a thread that uses a concurrency controller is to adhere to that concurrency controller's interface. We show that the correctness of a thread can be verified using stubs that are automatically generated from the concurrency controller interfaces. Such modularization of the verification task improves the efficiency of the verification and also helps us combine different approaches to verification with their associated strengths. We use an infinite-state specification checker (Action Language Verifier [BYK01]) for the verification of component behaviors. We use a program checker (JPF [BHPV00]) for the verification of the thread behavior. Since we use stubs in verification of the threads, the cost of thread verification drops drastically.

We demonstrate the effectiveness of the approach proposed in this paper on concurrency controllers for a bounded-buffer protected by a reader-writer lock, the sleeping barber problem, and an airport ground traffic control simulation program. Since we are using infinite state verification techniques we do not restrict integer variables to finite domains. Using counting abstraction we are also able to verify these controllers for arbitrary number of user threads.

Related Work: In [PDH99] the environment (i.e., the interface) of a software component is specified using LTL formulas. Then implementations of these environments are synthesized from the LTL specifications and combined with the components and verified using finite state model checkers. We use finite state machines to specify interfaces. The separation of thread and concurrency controller verification can be viewed as an assume-guarantee verification where the concurrency controller behavior is verified assuming that the user threads obey the interface. Similarly thread verification using stubs assumes that the behavior of the concurrency controller is verified independently.

We use interfaces to specify the order in which the methods of a concurrency controller are called whereas in [dAH01a, CdAHM02] interfaces can also be used to state input assumptions and output guarantees. In our approach, interface of a concurrency controller specifies only the calls to a concurrency controller, not the calls from a concurrency controller to other components. The approach presented in [dAH01a, CdAHM02] handles both calls to and from a component, and, hence, is more general. However, in our approach concurrency controllers which call other concurrency controllers can be modelled by composing individual concurrency controllers. Also, unlike the approach in [dAH01a, CdAHM02] our goal is to specify both the interface and the behavior of a component and verify them modularly using the behavior/interface separation.

A path expression specifies the synchronization constraints by describing the allowed concurrent execution sequences for a set of procedures [CH74]. In this sense, an interface specification is similar to a path expression. However, we use interface specifications to describe the behavior of a single thread as opposed to concurrent behavior of multiple threads. Hence, interface specifications do not have any synchronization constraints as in path expressions.

In [DDHM02], Deng et al. present an approach for synthesizing synchronization in concurrent programs from invariant specifications. We address verification of both synchronization primitives and their usage and our approach handles all ACTL properties not just invariants.

Our work in this paper extends the approach presented in [YKB02]. In [YKB02] thread verification is not addressed and concurrency controllers are specified directly in Action Language [Bul00]. The specification language we present in this paper is specialized for concurrency controllers. We automatically generate Action Language specifications and Java implementations from the concurrency controller specifications. By introducing interface specifications (which is not addressed in [YKB02]) we modularize both the specification and the verification of concurrency controllers.

The rest of the paper is organized as follows. In Section 2 we define the concurrency controllers, their behavior and interface specifications and their semantics. In Section 3 we discuss composition of concurrency controllers and define a refinement relation among interfaces which enables us to show that the properties of the composed concurrency controllers can be verified modularly. In Section 4 we discuss automated verification of concurrency controllers. In Section 5 we discuss generating Java implementations from concurrency controller specifications which preserve their properties. In Section 6 we present experimental results on automated verification of concurrency controllers. Finally, in Section 7 we give our conclusions.

2 Specification of Concurrency Controllers

A concurrency controller specification is a tuple $CC = (V, IC, RC, A, l, F)$ where V is a set of variables, IC is the initial condition, RC is the restrict condition, A is a set of actions, l is the synchronization lock, and F is the component interface. The variables in V can have the following types: *boolean*, *enumerated*, and *integer*. In the specification in Figure 1(a), the set of variables is $V = \{\texttt{nr}, \texttt{busy}\}$. The variable `nr` is of type *integer*, and the variable `busy` is of type *boolean*.

The initial condition IC and the restrict condition RC are predicates on the variables in V, i.e., $IC, RC : \prod_{v\in V} \text{DOM}(v) \rightarrow \{\text{TRUE}, \text{FALSE}\}$ where $\text{DOM}(v)$ denotes the domain of variable v and $\prod_{v\in V} \text{DOM}(v)$ denotes the Cartesian product of the variable domains. We restrict the predicates on integer variables to linear arithmetic. In the reader-writer example the initial condition of the controller is specified after the keyword `initial` as `nr=0 and !busy`. The restrict condition is used to restrict the state space of the system. In reader-writer concurrency controller the restrict condition is specified after the keyword `restrict` as `nr>=0` which means that `nr` is restricted to be nonnegative.

Behavior Specification: Behavior specification of a concurrency controller defines how the shared variables in that concurrency controller change their values. The variables of the concurrency controller can only be accessed and updated through the concurrency controller's actions.

The set of actions A specify 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. Actions are synchronized using the synchronization lock l. For each guarded command $gc \in a.GC$, guard $gc.g$ is a predicate on the variables V, such that, $gc.g : \prod_{v\in V} \text{DOM}(v) \rightarrow \{\text{TRUE}, \text{FALSE}\}$. For each guarded command $gc \in a.GC$, the update block $gc.u = (u_1, u_2, \ldots, u_k)$ is a sequence of update statements that are executed sequentially. Each update statement $u_i, 1 \leq i \leq k$, defines an update function $u_i : \prod_{v\in V} \text{DOM}(v) \rightarrow \prod_{v\in V} \text{DOM}(v)$.

In airport ground traffic control example given in Figure 2 the `exitRW3` action has one guarded command gc. The guard $gc.g$ is `numC3=0` and the update sequence $gc.u$ is (u_1, u_2) where u_1 is defined by the assignment `numC3:=numC3+1` and u_2 is defined by the assignment `numRW16R:=numRW16R-1`. Here, we will informally describe the semantics of the actions. When an action is called, if $l =$ TRUE, one guarded command whose guard evaluates to TRUE is arbitrarily chosen for execution. If $l =$ FALSE, the calling thread will wait until l becomes TRUE. If $l =$ TRUE and all the guards evaluate to FALSE, the behavior of the action depends on the blocking/nonblocking tag. The tag `blocking` means that the action has to execute a guarded command. I.e., if all guards evaluate to FALSE the calling thread should wait until some guard becomes TRUE. A waiting thread releases the synchronization lock, and re-acquires it before starting to execute an enabled guarded command. A nonblocking action does not cause the calling thread to wait. If the action is nonblocking and all of its guards evaluate to FALSE, a no-op command is executed.

In the example shown in Figure 1(a), the behavior is specified with four actions. The actions `r_enter` and `w_enter` are blocking actions. The actions `r_exit` and `w_exit` are nonblocking actions. These actions have no guards (which is equivalent to having `true` as a guard).

Figure 2 shows part of the behavior specification of a concurrency controller for an Airport Ground Traffic Control simulation program [Zho97, YKB02]. In this specification, the shared resources of the Airport Ground Traffic Control, runways and taxiways, are implemented as integer variables. For example, the variables `numRW16R` and `numC3` denote the number of airplanes on runway 16R and on taxiway C3, respectively. Behavior of the concurrency controller is defined using actions `reqLand`, `exitRW3`, and so on. The complete specification of this controller has 13 shared integer variables and 20 actions. These actions are called to simulate the behavior of an airplane in the airport ground network model similar to Seattle/Tacoma International Airport [Zho97].

Interface Specification: The interface specification F of a concurrency controller CC defines the acceptable call sequences for the threads that use CC. These allowed call sequences are specified using 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 and $RF : SF \times A \cup \{\epsilon\} \times SF$ is the transition relation. Note that, each transition in the interface specification corresponds to an action or an ϵ transition. In the interface specification of the reader-writer controller

```
ReaderWriter {
 integer nr;
 boolean busy;
 initial: nr=0 and !busy;
 restrict: nr>=0;
 blocking r_enter {[!busy] nr := nr+1;}
 nonblocking r_exit {[] nr := nr-1;}
 blocking w_enter {[nr=0 and !busy] busy := true;}
 nonblocking w_exit {[] busy := false;}
 interface {
   states: {idle,reading,writing}
   initial: idle
   (idle,r_enter,reading)
   (reading,r_exit,idle)
   (idle,w_enter,writing)
   (writing,w_exit,idle) }
}
```

(a)

```
ProducerConsumer {
  integer count;
  parameterized integer size;
  initial: count=0;
  restrict: size>0;
  nonblocking produce { [count<size] count := count+1; }
  nonblocking consume { [count>0] count := count-1; }
  interface {
    states: {init}
    initial: init
    (init,bproduce,init)
    (init,cconsume,init) }
    (init,produce,init)
    (init,consume,init) }
}
```

(b)

Figure 1: Reader-Writer and Producer-Consumer concurrency controllers

```
AirportGroundTrafficControl {
  integer numRW16R, numRW16L, numC3 ...;
  initial: numRW16R=0 and numRW16L=0 and numC3=0 ...;
  restrict: numRW16R>=0 and numRW16L>=0 and numC3>=0...;
  blocking reqLand { [numRW16R=0] numRW16R := numRW16R+1; }
  blocking exitRW3 { [numC3=0] numC3 := numC3+1; numRW16R := numRW16R-1; }
  blocking crossRW3 { [numRW16L=0 and numB2A=0] numC3 := numC3-1; numB2A := numB2A+1; }
  blocking reqTakeOff {
    [numRW16L=0 and numC3=0 and numC4=0 and numC5=0 and numC6=0 and numC7=0 and numC8=0]
      numRW16L := numRW16L+1;
  }
  blocking leave { [] numRW16L := numRW16L-1; } . . .
}
```

Figure 2: Airport Ground Traffic Control concurrency controller

given in Figure 1(a), the states in SF are listed as `idle`, `reading`, `writing`, and the IF is `idle`. The transitions in RT are listed as tuples, where, for example, `(idle,r_enter,reading)` denotes a transition from interface state `idle` to `reading` on action `r_enter`. In order to formalize the semantics, for each transition $t = (s_1, a, s_2)$ in RF where a is a blocking action, we add a wait state w_t to the interface states SF and we add two transitions (s_1, a, w_t) and (w_t, a, s_2) to the interface relation RF.

The interfaces of the concurrency controllers in Figures 1 and 2 are shown in Figure 3 (without the extra states and transitions added for blocking actions). The interface of the reader-writer concurrency controller indicates that a thread should execute the `r_exit` action only after a matching execution of the `r_enter` action. Similarly, a thread should execute the `w_exit` action only after a matching execution of the `w_enter` action. The producer-consumer concurrency controller interface specifies that a thread can execute produce and consume actions in arbitrary order. Figure 3(c) shows the interface of the concurrency controller for the Airport Ground Traffic Control simulation program. Figure 4 shows the interface specification of reader-writer controller after the addition of wait states and corresponding transitions.

Semantics: 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 user threads, ST is the set of states, $IT \subseteq ST$ is the set of initial states in the transition system, and $RT \subseteq ST \times ST$ is the transition relation.

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

We introduce the following notation. Given a state $s \in ST$ and a variable $v \in V$, $s(v) \in \text{DOM}(v)$ denotes the value of variable v in state s. Similarly, $s(l) \in \{\text{TRUE}, \text{FALSE}\}$ denotes the value of lock l in s. Given a state $s \in ST$ and a set of variables $V' \subseteq V$, $s(V') \in \prod_{v \in V'}$ denotes the projection of state s to the domains of the variables in V'. Similarly, $s(SF) \in \prod^n SF$ denotes the projection of state s to the states of the threads. Given a state s and a thread t, where $1 \le t \le n$, $s(SF)(t) \in SF$ denotes the state of thread t in s and $s(SF - t) \in \prod^{n-1} SF$ denotes the projection of state s to the states of all the threads except thread t.

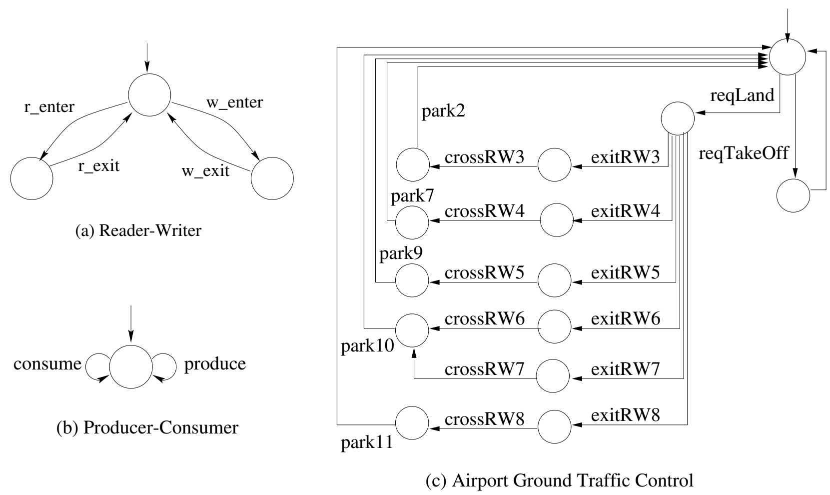


Figure 3: Concurrency controller interfaces

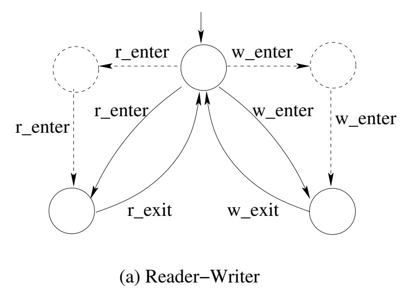


Figure 4: Reader-Writer interface with waiting states

Finally, given an action a, $blocking(a) \in \{\text{TRUE}, \text{FALSE}\}$ denotes if the action a is blocking or not, and given an interface state $q \in SF$, $wait(q) \in \{\text{TRUE}, \text{FALSE}\}$ denotes if the interface state q is a wait state or not.

The initial states of the transition system $T(CC)(n)$ is defined as $IT = \{s \mid s \in ST \wedge IC(s) \wedge s(l) \wedge \forall 1 \leq t \leq n,\ s(SF)(t) = IF\}$. To define RT we will also define two functions ACQ and REL. The $\text{ACQ}(s)(l)$ function returns a state s' such that $s'(l) = \text{FALSE}$, $s'(V) = s(V)$ and $s'(SF) = s(SF)$, i.e., s' is equivalent to state s except that the lock is taken in s'. Similarly, the $\text{REL}(s)(l)$ function returns a state s' such that $s'(l) = \text{TRUE}$, $s'(V) = s(V)$ and $s'(SF) = s(SF)$.

We define $RT^u_{(q,a,q')}$, the transition relation for a transition (q, a, q') when one of its guarded commands is executed as

$$
\begin{aligned}
RT^u_{(q,a,q')} = \{&(s_0, s_1), \ldots, (s_{k+1}, s_{k+2}) \mid (\forall 0 \leq i \leq k+2,\ s_i \in ST) \ \wedge\ s_0(l) \ \wedge\ s_1 = \text{ACQ}(s_0)(l) \\
&\wedge (\exists 1 \leq t \leq n, s_{k+1}(SF)(t) = q' \ \wedge\ (\forall 0 \leq i < k+1,\ s_i(SF)(t) = q \ \wedge\ s_{i+1}(SF - t) = s_i(SF - t))) \\
&\wedge (\exists gc \in a.GC,\ gc.g(s_0) \ \wedge\ gc.u = (u_1, u_2, \ldots, u_k) \ \wedge (\forall 1 \leq i \leq k,\ s_{i+1}(V) = u_i(s_i(V)) \\
&\wedge s_{i+1}(l) = s_i(l))) \ \wedge\ s_{k+2} = \text{REL}(s_{k+1})(l)\}
\end{aligned}
$$

The transition relation for a nonblocking action a is $RT_{(q,a,q')} = RT^u_{(q,a,q')} \cup RT^{nb}_{(q,a,q')}$ where $RT^{nb}_{(q,a,q')}$ denotes the case where none of the guards of a evaluate to TRUE

$$
\begin{aligned}
RT^{nb}_{(q,a,q')} = \{&(s_0, s_1) \mid s_0, s_1 \in ST \ \wedge\ s_0(l) \ \wedge\ s_1(l) \ \wedge\ s_1(V) = s_0(V) \ \wedge\ (\forall gc \in a.GC,\ \neg gc.g(s_0)) \\
&\wedge (\exists 1 \leq t \leq n,\ s_0(SF)(t) = q \ \wedge\ s_1(SF)(t) = q' \ \wedge\ s_1(SF - t) = s_0(SF - t))\}
\end{aligned}
$$

For a blocking action a, the transition relation is $RT_{(q,a,q')} = RT^u_{(q,a,q')} \cup RT^b_{(q,a,w)} \cup RT^u_{(w,a,q')}$ where

$$
\begin{aligned}
RT^b_{(q,a,w)} = \{&(s_0, s_1) \mid s_0, s_1 \in ST \ \wedge\ s_0(l) \ \wedge\ s_1(l) \ \wedge\ s_1(V) = s_0(V) \ \wedge\ (\forall gc \in a.GC,\ \neg gc.g(s)) \\
&\wedge (\exists 1 \leq t \leq n,\ s_0(SF)(t) = q \ \wedge\ s_1(SF)(t) = w \ \wedge\ s_1(SF - t) = s_0(SF - t))\}
\end{aligned}
$$

and w is the wait state for the transition (q, a, q'). Finally,

$$RT_{(q,\epsilon,q')} = \{(s_0, s_1) \mid s_0, s_1 \in ST \ \wedge \ s_0(l) \ \wedge \ s_1(l) \ \wedge \ s_1(V) = s_0(V) \\ \wedge \ (\exists 1 \le t \le n, \ s_0(SF)(t) = q \ \wedge \ s_1(SF)(t) = q' \ \wedge \ s_1(SF - t) = s_0(SF - t))\}$$

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 and the observable paths of the transition system $T(CC)(n)$ as follows: An execution path $s_0, s_1, \ldots$ is a path such that $s_0 \in IT$ and $\forall i \ge 0, \ (s_i, s_{i+1}) \in RT$. An observable state s is a state in ST where $s(l) = \text{TRUE}$. An observable path $s_0, s_1, \ldots$ is the sequence of observable states in an execution path (i.e., an observable path is a subsequence of an execution path and it includes all the observable states in the execution path).

Let AP denote the set of atomic properties, where a property $p \in AP$ is a predicate on variables in V, $p : \prod_{v \in V} \text{DOM}(v) \to \{\text{TRUE}, \text{FALSE}\}$. We use CTL to state properties of the transition system $T(CC)(n)$. A concurrency controller CC satisfies a CTL formula f, if and only if, $\forall n \ge 0$, all the initial states of the transition system $T(CC)(n)$ satisfy the formula f on observable paths, i.e., the CTL semantics for concurrency controllers are defined on observable paths (instead of execution paths).

3 Composition of Concurrency Controllers

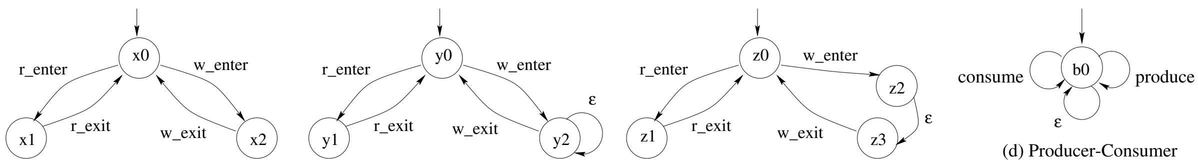
Interfaces of different concurrency controllers can be composed to form more complex concurrency controllers. In an interface composition, actions of different concurrency controllers can be interleaved or can be combined and executed simultaneously (we call this synchronous composition).

Let $CC_1, CC_2, \ldots, CC_m$ be m concurrency controller specifications with disjoint variables, actions and locks, such that $CC_i = (V_i, \ IC_i, \ RC_i, \ A_i, \ l_i, \ F_i)$ and $F_i = (IF_i, SF_i, RF_i)$ for $1 \le i \le k$. Let $CC_c = (V_c, \ IC_c, \ RC_c, \ A_c, \ L_c, \ F_c)$ be the composed concurrency controller where $V_c = \bigcup_{1 \le i \le m} V_i$, $IC_c = \bigwedge_{1 \le i \le m} IC_i$, $RC_c = \bigwedge_{1 \le i \le m} RC_i$, $A_c = \bigcup_{1 \le i \le m} A_i$, and $L_c = \{l_1, l_2, \ldots, l_m\}$. A composed controller has a set of locks, one for each individual concurrency controller. The state space of a composed concurrency controller is defined similar to individual controllers by taking the Cartesian product of all the variables, locks and the composed interface. The guards and the updates defining the actions of individual concurrency controllers are extended to the Cartesian product of the domains of all the variables in the composed component in a straightforward way: an update for an individual concurrency controller preserves the values of the variables of the other concurrency controllers. Note that, all parts of a composed concurrency controller other than its interface F_c is determined by the individual concurrency controllers that are composed.

Let the interface of the composed concurrency controller be $F_c = (IF_c, \ SF_c, \ RF_c)$. Given $A_c = \bigcup_{1 \le i \le m} A_i$, the set of composed actions $CA \subseteq 2^{A_c}$ is defined as follows: For each composed action $ca = \{a_1, a_2, \ldots, a_r\} \in CA$, for each A_i, $|ca \cap A_i| \le 1$ and $r \ge 1$, i.e., each composed action ca contains at most one action from each A_i. Then the transition relation of the composed concurrency controller interface is defined as $RF_c : SF_c \times CA \times SF_c$. If a transition of a composed interface is labelled by a singleton set, then that transition corresponds to executing a single action from an individual component. On the other hand, a transition which is labelled by more than one action corresponds to executing multiple actions from different concurrency controllers synchronously. Note that a composed action can have a mixed set of blocking and nonblocking actions. A thread executing a composed action has to wait until all the blocking actions become executable.

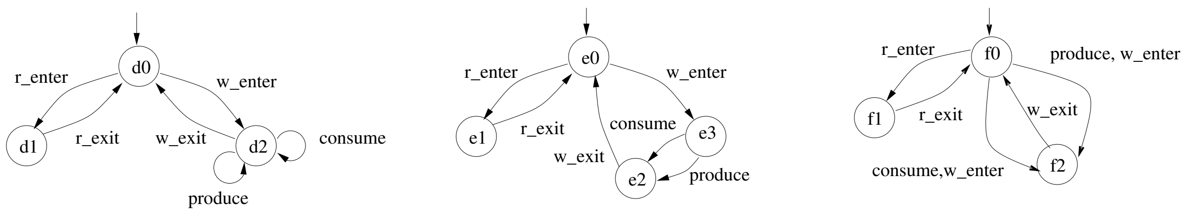
Let $T(CC_c)(n) = (IT, ST, RT)$ be the transition system of the composed concurrency controller $CC_c = (V_c \ , IC_c, \ RC_c, \ A_c, \ L_c, \ F_c)$. The initial states IT and the set of states ST of the transition system $T(CC_c)(n)$ is based on V_c, IC_c, RC_c, and L_c similar to the semantics of individual concurrency controllers discussed above. To define the transition relation RT of a composed concurrency controller we need to define the semantics for the transitions of the form $(q, ca, q') \in RF_c$ where $ca \in CA$. We do this by defining a set of composed update sequences for each composed action. The composed update sequences correspond to synchronous execution of all the individual actions in the corresponding composed action. Given an action a_i, let $\text{IND}(a_i) = j$ if $a_i \in A_j$ (a_i is an action from the individual component CC_j). Given a state $s \in ST$, and a composed action $ca = \{a_1, a_2, \ldots, a_r\} \in CA$, $\text{COMP}(ca)(s)$ is defined as follows:

- If there exists a blocking action $a_i \in ca$ such that for all guarded commands $gc \in a_i.GC$, $gc.g(s)$ is FALSE, then $\text{COMP}(ca)(s) = \emptyset$.



(a) Reader-Writer-1 (b) Reader-Writer-2 (c) Reader-Writer-3 (d) Producer-Consumer

Figure 5: Individual interfaces



(a) (b) (c)

Figure 6: Composed interfaces

- Otherwise, for all actions $a_i \in ca = \{a_1, a_2, \ldots, a_r\}$ choose a guarded command $gc_i \in a_i.GC$ such that $gc_i.g(s)$ is true. If there is no such guarded command for an action a_i (which implies that a_i is non-blocking) let gc_i be gc_I, a guarded command with a TRUE guard and an update sequence of length 0. The composed update sequence u^c, corresponding to composition of these guarded commands, is defined as $u^c = (u^c_1, u^c_2, \ldots, u^c_k)$ where either k is the length of the longest update sequence among $gc_1.u, gc_2.u, \ldots, gc_r.u$ or $k = 1$ and the length of the longest update sequence is 0. Each step u^c_j in the update sequence u^c is defined as follows: $s' = u^c_j(s)$ if and only if for all $1 \le i \le r$, $s'(V_{\mathrm{IND}(a_i)}) = gc_i.u_j(s(V_{\mathrm{IND}(a_i)}))$. If j is greater than the length of the update sequence $gc_i.u$ then $gc_i.u_j$ is the identity function. $\mathrm{COMP}(ca)(s)$ is the set of all such composed update sequences.

Given a transition (q, ca, q') we define the following:

$$\begin{aligned}
RT^u_{(q,ca,q')} &= \{(s_0, s_1), \ldots, (s_{k+1}, s_{k+2}) \mid (\forall 0 \le i \le k+2,\ s_i \in ST) \ \wedge\ (\forall 1 \le i \le m, s_0(l_i)) \\
&\wedge\ s_1 = \mathrm{ACQ}(s_0)(l_1, l_2, \ldots, l_m) \ \wedge\ s_{k+2} = \mathrm{REL}(s_{k+1})(l_1, l_2, \ldots, l_m) \\
&\wedge\ (\exists 1 \le t \le n,\ s_{k+1}(SF)(t) = q' \ \wedge\ (\forall 0 \le i < k+1,\ s_i(SF)(t) = q \ \wedge\ s_{i+1}(SF - t) = s_i(SF - t))) \\
&\wedge\ (\exists u^c = (u^c_1, u^c_2, \ldots, u^c_k) \in \mathrm{COMP}(ca)(s_0), (\forall 1 \le i \le k,\ s_{i+1}(V_c) = u^c_j(s_i(V_c)))\}
\end{aligned}$$

Given a transition (q, ca, q'), if none of the actions in ca are blocking, then $RT_{(q,ca,q')} = RT^u_{(q,ca,q')}$. If there is a blocking action in the composition then $RT_{(q,ca,q')} = RT^u_{(q,ca,q')} \cup RT^b_{(q,ca',w)} \cup RT^u_{(w,ca,q')}$ where w is the wait state for the transition (q, ca, q'), ca' is the set of blocking actions in ca, and

$$\begin{aligned}
RT^b_{(q,ca',w)} &= \{(s_0, s_1) \mid s_0, s_1 \in ST \ \wedge\ (\forall 1 \le i \le m,\ s_0(l_i) \ \wedge\ s_1(l_i)) \\
&\wedge\ (\exists 1 \le t \le n,\ s_0(SF)(t) = q \ \wedge\ s_1(SF)(t) = q' \ \wedge\ s_1(SF - t) = s_0(SF - t)) \\
&\wedge\ \mathrm{COMP}(ca')(s_0) = \emptyset \ \wedge\ (\forall 1 \le i \le m,\ s_1(V_i) = s_0(V_i))\}
\end{aligned}$$

Finally, the overall transition relation RT of $T(CC_c)(n)$ is: $RT = \bigcup_{(q,ca,q') \in RF} RT_{(q,ca,q')}$.

The reader-writer and the producer-consumer concurrency controllers can be composed in several different ways. Three of these compositions are given in Figure 6(a), 6(b) and 6(c). In Figure 6(a), when a thread is writing, it could execute arbitrary number of `produce` and `consume` actions. In the interface given in Figure 6(b), however, when a thread is writing, it should execute either one `produce` or one `consume` action before it exits writing. There are two synchronously composed actions in Figure 6(c), one of them is the synchronous composition of `produce` and `w_enter` actions and the other one is the synchronous composition of `consume` and `w_enter` actions. Figure 7 shows the same interface as in Figure 6(c) after the addition of wait states and corresponding transitions for the blocking actions.

Refinement Relation: Here we define a refinement relation for interfaces. The idea is that if a composed interface is a refinement of another interface then the ACTL properties verified on the original controller is preserved by the composed controller.

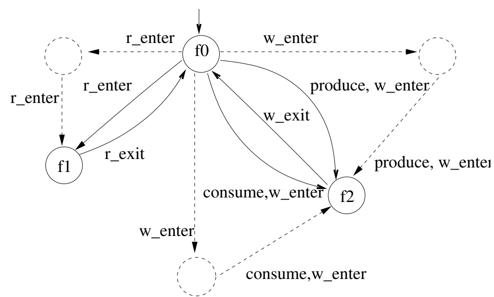
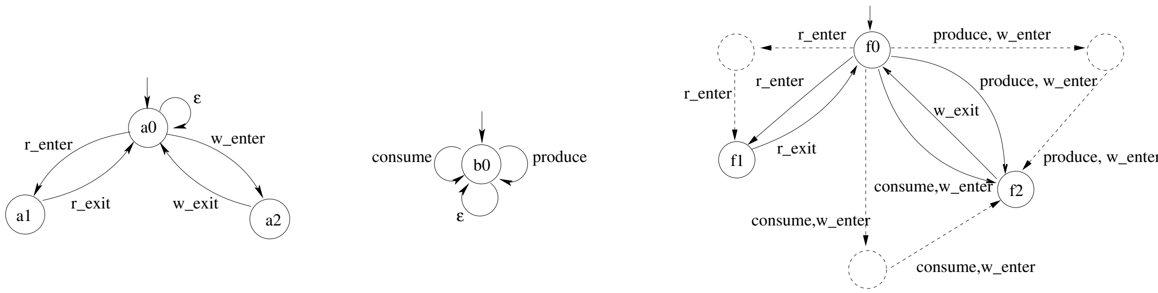


Figure 7: Composed interface with wait states (produce and consume actions are nonblocking)



(a) Reader–Writer (b) Producer–Consumer (c) Composed Interface

Figure 8: Composition of Reader-Writer and Producer-Consumer controllers with blocking produce and blocking consume actions

Let $CC_1, CC_2, \ldots, CC_m$, be m concurrency controller specifications, and let $CC_c = (V_c,\ IC_c,\ RC_c,\ A_c,\ L_c,\ F_c)$ be a composition of these concurrency controllers and let $F_c = (\ IF_c,\ SF_c,\ RF_c)$ be the composition interface. We use $F_c \preceq F_i$ to denote that the composed interface F_c is a refinement of the interface F_i.

Definition 3.1 *$F_c \preceq F_i$ if and only if there exists a mapping $H : SF_c \rightarrow SF_i$ such that for all $q_c \in SF_c$ and for all $q \in SF_i, H(q_c) = q$ implies that, $q_c \in IF_c \Rightarrow q \in IF_i$ and for each $((q_c, ca, q'_c) \in RF_c$*

$$
\begin{array}{l}
(a \in ca \cap A_i \Rightarrow \exists (q, a, q') \in RF_i, H(q'_c) = q') \\
\wedge\ (ca \cap A_i = \emptyset \Rightarrow \exists (q, \epsilon, q') \in RF_i, H(q'_c) = q') \\
\wedge\ (ca \cap A_i \neq \emptyset\ \wedge \neg wait(q)\ \wedge\ (\exists A_j\ i \neq j,\ b \in ca \cap A_j\ \wedge\ blocking(b)) \Rightarrow \exists (q, \epsilon, q) \in RF_i,)
\end{array}
$$

In Figure 6 we give three different composed interfaces and in Figure 5 we give three different reader-writer interfaces. Note that not all of the composed interfaces refine all the reader-writer interfaces. The interface in Figure 6(a) is a refinement of reader-writer interface in Figure 5(b). Figure 6(b) is a refinement of reader-writer interface in Figure 5(b) and (c). Figure 6(c) is a refinement of reader-writer interface in Figure 5(a) and (b).

To illustrate synchronous composition of two blocking actions, we describe another composition in Figure 8. In this example, there are two individual concurrency controllers. First one is a Reader-Writer controller with the interface shown in Figure 8(a), which has an ϵ-transition in its initial state. The other one is a Producer-Consumer controller with the interface shown in Figure 8(b). In this example, we assume that the `produce` and the `consume` actions are blocking actions. A composition of these two interfaces is shown in Figure 8(c) including the wait states.

Let CC_i be one of the controllers in the composition of the composed controller CC_c. Let $T(CC_c)(n) = (IT_c,\ ST_c,\ RT_c)$ be the transition system for CC_c and $T(CC_i)(n) = (IT_i,\ ST_i,\ RT_i)$ be the transition system for CC_i. Given a mapping $H : SF_c \rightarrow SF_i$ between the interface states of the CC_c and CC_i we define a projection function $\Pi : ST_c \rightarrow ST_i$ such that, given $s_c \in ST_c$, $\Pi(s_c)(V_i) = s_c(V_i)$ and $\Pi(s_c)(l_i) = s_c(l_i)$ and for all $1 \leq t \leq n$, $\Pi(s_c)(SF_i)(t) = H(s_c(SF)(t))$. Observe that, for any atomic property $p \in AP_i$,

$s_c \models p$ if and only if $\Pi(s_c) \models p$. We can generalize the projection function to paths as follows: Given a path $\sigma^c = \sigma_0^c, \sigma_1^c, \ldots$ in CC_c, we define $\Pi(\sigma^c)$ as $\Pi(\sigma^c) = \Pi(\sigma_0^c), \Pi(\sigma_1^c), \ldots$.

Lemma 3.2 *If $F_c \preceq F_i$, then for all observable paths σ of CC_c, the projection $\Pi(\sigma)$ is an observable path in CC_i where projection Π is based on the mapping function H from Definition 3.1 that shows that $F_c \preceq F_i$.*

Let $\sigma = s_0^c, s_1^c, \ldots$ be an observable path in CC_c. We show by induction that, for each prefix $s_0^c, s_1^c, \ldots, s_n^c$ of the path σ, $\Pi(s_0^c), \Pi(s_1^c), \ldots, \Pi(s_n^c)$ is the prefix of an observable path in CC_i. The base case is obvious since $\Pi(s_0^c)$ is the prefix of any observable path starting from $\Pi(s_0^c)$. We assume that $\Pi(s_0^c)$, $\Pi(s_1^c)$, ..., $\Pi(s_n^c)$ is the prefix of an observable path and show that $\Pi(s_0^c), \Pi(s_1^c), \ldots, \Pi(s_n^c), \Pi(s_{n+1}^c)$ is the prefix of an observable path.

Let $\{(s_n^c, s_{n_1}^c), (s_{n_1}^c, s_{n_2}^c), \ldots, (s_{n_k}^c, s_{n+1}^c)\} \subseteq RT_{(q,ca,q')} \subseteq RT_c$, where $s_{n_1}^c, \ldots, s_{n_k}^c$ are not observable states. We will show that, $\{(\Pi(s_n^c), \Pi(s_{n_1}^c)), (\Pi(s_{n_1}^c), \Pi(s_{n_2}^c)), \ldots, (\Pi(s_{n_l}^c), \Pi(s_{n+1}^c))\} \subseteq RT_i$ for some l, where $0 \leq l \leq k$ and $\Pi(s_{n_1}^c) = \Pi(s_{n_{l+1}}^c) = \ldots = \Pi(s_{n_k}^c)$ and $\Pi(s_{n_1}^c), \ldots, \Pi(s_{n_l}^c)$ are not observable states. There are two cases two consider:

1. $\exists a \in ca,\ a \in A_i$:
   We know that $(H(q), a, H(q')) \in SF_i$ since $F_c \preceq F_i$. There are two cases:
   (a) $\{(s_n^c, s_{n_1}^c), (s_{n_1}^c, s_{n_2}^c), \ldots, (s_{n_k}^c, s_{n+1}^c)\} \subseteq RT^u_{(q,ca,q')}$
   There are three cases to consider:
      i. a is a nonblocking action and for all $gc \in a.GC$, $gc.g(\Pi(s_n^c))$ is FALSE.
      By definition of the composed update sequence (COMP), defined earlier in this section, the guarded command for action a in the composed update sequence is gc_I, which has an update sequence of length 0. Therefore, $\Pi(s_{n_1}^c) = \Pi(s_{n_2}^c) = \ldots = \Pi(s_{n_k}^c)$, and we choose $l = 0$. Hence, we just need to show that $\{(\Pi(s_n^c), \Pi(s_{n+1}^c))\} \subseteq RT_i$. Since the length of the update sequence for gc_I is 0, $\Pi(s_n^c) = \Pi(s_{n+1}^c)$ by definition of the $RT^u_{(q,ca,q')}$. Since, $\Pi(s_n^c) = \Pi(s_{n+1}^c)$ and $(H(q), a, H(q')) \in RF_i$, and a is a nonblocking action, $(\Pi(s_n^c), \Pi(s_{n+1}^c)) \in RT^{nb}_{(H(q),a,H(q'))} \subseteq RT_{(H(q),a,H(q'))} \subseteq RT_i$.
      ii. a is a nonblocking action and there exists a $gc \in a.GC$, $gc.g(\Pi(s_n^c))$ is TRUE.
      We can assume that the guarded command for action a in the composed update sequence is gc with an update sequence of length l. By definition of the composed update sequence, $\Pi(s_{n_l}^c) = \Pi(s_{n_{l+1}}^c) = \ldots = \Pi(s_{n_k}^c)$. By definition of $RT^u_{(H(q),a,H(q'))}$, we can show that, $\{(\Pi(s_n^c), \Pi(s_{n_1}^c)), (\Pi(s_{n_1}^c), \Pi(s_{n_2}^c)), \ldots, (\Pi(s_{n_l}^c), \Pi(s_{n+1}^c))\} \subseteq RT^u_{(H(q),a,H(q'))} \subseteq RT_{(H(q),a,H(q'))}$ $\subseteq RT_i$ where $\Pi(s_{n_1}^c), \Pi(s_{n_2}^c), \ldots, \Pi(s_{n_l}^c)$ are not observable states.
      iii. a is a blocking action.
      By definition of $RT^u_{(q,ca,q')}$, COMP(ca) cannot be empty in this case. Since COMP(ca) is not empty, there should exists a $gc \in a.GC$ where $gc.g(s_n^c) =$ TRUE. Then, with a reasoning similar to the case (ii) above, we can show that $\{(\Pi(s_n^c), \Pi(s_{n_1}^c))$, $(\Pi(s_{n_1}^c), \Pi(s_{n_2}^c))$, $\ldots, (\Pi(s_{n_l}^c), \Pi(s_{n+1}^c))\} \subseteq RT^u_{(H(q),a,H(q'))} \subseteq RT_{(H(q),a,H(q'))} \subseteq RT_i$ where $l \leq k$ and $\Pi(s_{n_1}^c)$, $\Pi(s_{n_2}^c), \ldots, \Pi(s_{n_l}^c)$ are not observable states and $\Pi(s_{n_l}^c) = \Pi(s_{n_{l+1}}^c) = \ldots = \Pi(s_{n_k}^c)$.
   (b) $\{(s_n^c, s_{n+1}^c)\} \subseteq RT^b_{(q,ca,q')} \ \wedge\ wait(q')$
   In this case, by definition of the composed transition relation, ca contains only blocking actions and COMP$(ca) = \emptyset$, i.e., there exists a blocking action $b \in ca$ such that for all guarded commands $gc \in b.GC, gc.g(s_n^c)$ is FALSE. There are two cases to consider:
      i. a is a blocking action and for all $gc \in a.GC$, $gc.g(\Pi(s_n^c))$ is FALSE.
      Since COMP$(ca) = \emptyset$, $s_n^c(V) = s_{n+1}^c(V)$. Therefore, the corresponding transition in RT_i is $(\Pi(s_n^c), \Pi(s_{n+1}^c))\} \subseteq RT^b_{(H(q),a,H(q'))} \subseteq RT_{(H(q),a,H(q'))} \subseteq RT_i$ where $H(q')$ is a wait state in ST_i.
      ii. a is a blocking action and there exists a $gc \in a.GC$, $gc.g(\Pi(s_n^c))$ is TRUE.
      Since there is another blocking action in the composition, by definition of the refiment relation, $(H(q), \epsilon, H(q)) \in RF_i$. Therefore, the corresponding transition in RT_i is $\{(\Pi(s_n^c), \Pi(s_{n+1}^c))\}$ $\subseteq RT_{(H(q),\epsilon,H(q'))} \subseteq RT_i$ where $H(q) = H(q')$.

2. $\forall a \in ca, a \notin A_i$:
We know that $(H(q), \epsilon, H(q')) \in RF_i$ since $F_c \preceq F_i$. Since the only way to modify the valuations of variables in V_i is through actions in A_i and no action of A_i is in the ca, $s^c_{n+1}(V_i) = s^c_n(V_i)$ and $\Pi(s^c_n) = \Pi(s^c_{n_1}) = \Pi(s^c_{n_2}) = \ldots = \Pi(s^c_{n_k})$. Therefore, the corresponding transition is $\{(\Pi(s^c_n), \Pi(s^c_{n+1}))\} \in RT_{(H(q),\epsilon,H(q'))} \subseteq RT_i$ where $l = 0$.

Hence, in either case, $\Pi(s^c_0), \Pi(s^c_1), \ldots, \Pi(s^c_n), \Pi(s^c_{n+1})$ is the prefix of an observable path in CC_i.

Based on the above lemma we can prove the following theorem:

Theorem 3.3 *Given a concurrency controller $CC_c = (V_c, IC_c, RC_c, A_c, L_c, F_c)$ which is a composition of concurrency controllers $CC_1, CC_2, \ldots, CC_m$, CC_c preserves all ACTL properties of CC_i, $1 \leq i \leq m$, if $F_c \preceq F_i$.*

Let Π denote the projection based on the mapping function H from Definition 3.1 that shows that $F_c \preceq F_i$. In order to prove that the ACTL properties of CC_i are preserved in CC_c it is sufficient to show that for all $s \in ST_c$, $\Pi(s) \models \mathrm{A}f \Rightarrow s \models \mathrm{A}f$ assuming that f is of the form $\mathrm{F}p$, $\mathrm{G}p$, $p\mathrm{U}q$, and $\Pi(s) \models p \Rightarrow s \models p$ and $\Pi(s) \models q \Rightarrow s \models q$. Note that $\Pi(s) \models p \Rightarrow s \models p$ when p is an atomic property of the concurrency controller CC_i. If we can prove the claim above, we can show by structural induction that Theorem 3.3 holds.

We are going to prove the above claim using proof by contradiction. Suppose the claim is not true, i.e. $\Pi(s) \models \mathrm{A}f \wedge s \not\models \mathrm{A}f$. Note that, $s \not\models \mathrm{A}f$ implies that $s \models \mathrm{E}\neg f$ which means that there exists a path σ in CC_c starting at s which satisfies $\neg f$. Then, according to the Lemma 3.2, there exists a path $\Pi(\sigma)$ in CC_i starting at $\Pi(s)$ which satisfies $\mathrm{E}\neg f$ which is a contradiction.

Using Theorem 3.3, we are able to verify the ACTL properties of the individual concurrency controllers modularly before composing them. If we can show that the refinement relation holds, then the verified properties are preserved in the composed system. However, the above theorem cannot be used for properties that refer to variables of different individual concurrency controllers. Such a property can be verified on the transition system of the composed concurrency controller CC_m after the composition.

4 Automated Verification of Concurrency Controllers

We verify the behavior of a concurrency controller (or a composed concurrency controller) using the Action Language Verifier. For this purpose, the specification of a concurrency controller CC is translated into the input language of the Action Language Verifier [YKB02]. Action Language Verifier is an infinite state symbolic model checker. It uses the Composite Symbolic Library [YKTB01] to encode the transition system $T(CC)(n) = (IT, ST, RT)$ of a concurrency controller (with n user threads) symbolically. In Action Language unspecified integer constants can be defined as parameterized constants and the verification is performed for every possible value of the parameterized constant. Given a transition system $T(CC)(n) = (IT, ST, RT)$ of a concurrency controller CC and a CTL property p, Action Language Verifier uses conservative approximation techniques such as widening and truncated fixpoint computations to conservatively verify or falsify infinite state specifications [BYK01, BGP99].

Counting Abstraction: We use an automated abstraction technique called counting abstraction [Del00] to verify the behavior of a concurrency controller (or a composed one) for arbitrary number of threads. Implementation of counting abstraction for Action Language specifications is discussed in [YKB02]. 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.

Thread Verification: Client threads which use a concurrency controller must adhere to the interface specification of the concurrency controller. Assume that the sequence of action executions by a client thread is $cs = a_0, a_1, a_2, \ldots$. A thread is correct with respect to an interface if the call sequence cs generated by the thread can also be generated by the finite state machine defining the interface. Note that, this actually corresponds to the refinement relation we defined for the interfaces in Section 3. In other words a thread is correct with respect to an interface if it is a refinement of that interface. If a thread implementation is a refinement of a concurrency controller's interface then the ACTL properties verified on the concurrency controller specifications are preserved.

```
public class Action{
  protected final Object CondVar, owner;
  private final Vector gcV, notifV;
  private boolean blocking=true;

  public Action(Object c, Vector gcs,boolean tag){...}
  void addNotification(Object n){...}
  private boolean GuardedExecute(){
    boolean result=false;
    synchronized(owner){
      for(int i=0;i<gcV.size(); i++)
       try{
        if(((GuardedCommand)gcV.get(i)).guard()){
          ((GuardedCommand)gcV.get(i)).updates();
           result=true;
           break; }
        } catch(Exception e){}
    }
    return result;
  }
```

```
public boolean nonblocking(){
  boolean result=GuardedExecute();
  if (result) notification();
  return result;
}
public void blocking(){
  synchronized(CondVar){
     while(!GuardedExecute())
        try{CondVar.wait();}
        catch (Exception e){}
  }
  notification();
}
public boolean enabled(){...}
protected void notification(){
  for(int i=0;i<notifV.size();i++)
    try{
      synchronized(notifV.get(i)){
        notifV.get(i).notifyAll();}
    } catch(Exception e){}
} }
```

Figure 9: Action Class

We used JPF [BHPV00] to check the correctness of client threads. JPF is a finite state program verification tool, which enables us to verify arbitrary Java threads without any restrictions. JPF supports property specifications via assertions that are embedded in the source code. To verify client threads we synthesize a stub class from concurrency controller interfaces to improve the efficiency of the thread verification. In the next section we will discuss the code generation for the stubs.

5 Code Generation from Specifications

Given a concurrency controller specification $CC = (V, IC, RC, A, l, F)$, we automatically synthesize a Java class that implements CC. We have implemented the Java counterpart of the action structure (Figure 9).The condition variable and the notification vector are used for implementing the specific notification pattern [Car96]. When a client thread is blocked while executing a blocking action, it waits on the condition variable of that action. Using a different condition variable for each blocked action improves the performance by awakening only the related threads. We compute the notification dependencies and create a notification list for each action [YKB02].

The Action class has three significant methods. The `GuardedExecute` method is for executing one of the guarded commands of the action. The method first acquires the controller's lock l, and then executes the updates of the first guarded command whose guard is true. If all the guards evaluate to false, then the 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 `GuardedExecute` method does not execute one of the guarded commands, then the thread waits, in a loop, on the action's condition variable until it is notified by another thread. The execution of a nonblocking action is implemented by the `nonblocking` method. This method calls `GuardedExecute` and does the necessary notifications. Since a nonblocking action does not cause the calling thread to wait, there is no *wait* statement in this method. Other than these methods, there is an `enabled()` method, which returns `true` if there is a $gc \in a.GC$ where $gc.g$ evaluates to `true`. If the action is nonblocking, this method always returns `true`. This method is necessary to implement synchronous compositions.

Below is an excerpt from the generated Java code for the Reader-Writer example. The shared variables `nr` and `busy` are declared as private fields. The generated class has one public method for each action. These methods have calls to the `blocking` or `nonblocking` method of the action instance. In the ReaderWriter class the `public void r_enter()` method is an example of such methods.

```
public class ReaderWriter {
   private int nr; private boolean busy;
   final Action r_enterAction; final Action r_exitAction; ...
   public ReaderWriter(){
      nr=0; busy=false; Vector gcs; gcs=new Vector();
      gcs.add(new GuardedCommand(){
```

```
            public boolean guard(){
                boolean result=false;
                synchronized(ReaderWriter.this){ result=!busy; }
                return result;}
            public void updates(){synchronized(ReaderWriter.this){ nr=nr+1;}} });
    r_enterAction=new Action(this,gcs,true); ...
    try{ r_exitAction.addNotification(w_enterAction.CondVar); ... }catch(Exception e){}
}
public void r_enter(){ r_enterAction.blocking();}...  }
```

The constructor method contains the code for the initializations of the shared variables and the action instances. Initial values of the variables in V are assigned based on the initial condition IC. The initialization of action instances in the generated constructor method is as follows: For each guarded command gc of the action a, an inner class implementing the GuardedCommand interface is generated. The GuardedCommand interface has two methods: a `guard()` method that has the implementation of $gc.g$ and an `update()` method that has the implementation of $gc.u$. We use inner classes so that the `update()` and the `guard()` methods can access the variables in V which are declared as private fields in the controller class. An unnamed instance of this class is added to a vector of GuardedCommands. The constructor of the Action class is invoked with this vector. After the action initializations, each action's notification vector is constructed with the condition variables of the related actions according to precomputed notification list.

We are also able to generate an optimized Java class by generating separate methods for execution of each action within the controller class, i.e., without using the Action class and avoiding the use of inner classes.

Composition: We have implemented a ComposedAction class as the Java counterpart of the synchronously composed action structure. This class extends the Action class (described above). The synchronous execution of all participant actions without interruption is implemented by a `GuardedExecute` method. This method first acquires the locks of each controller $CC_1.l$, $CC_2.l$, $\ldots$, $CC_m.l$ in order. If the composed action is enabled, then every action in the composition executes without releasing the concurrency controllers' locks. Otherwise, `false` is returned. If the `GuardedExecute` method returns false, the client thread waits on the condition variable of the composed action.

The generated composed concurrency controller class has a vector of controllers. For each action a which is not synchronously composed, a public method is generated in the composed concurrency controller. This method calls the corresponding method of the individual concurrency controller instance CC_i where $a \in CC_i.A$ after acquiring all controllers' locks. For each synchronously composed action, a ComposedAction instance is created. The generated composed concurrency controller class has one public method for each composed action.

Stub Generation: At the code generation phase, in addition to the controller class, we also generate a stub class which implements the interface $F = (IF, SF, RF)$ of the concurrency controller CC. This stub class is used in client thread verification. The stub has only one variable which is an integer keeping the current state of the thread. For each state $s \in SF$ an integer constant is declared as `final static int`. The constructor initializes the state variable to the state IF. For each action $a \in A$ a public method is generated. This method asserts that there exists a $t_i = (s_i, a, s'_i) \in RF$ such that s_i is current state. In stub class generation we do not consider the wait states, i.e. s'_i is not a wait state. The generated method then sets the current state to s'_i. Here, we assume that the transition system is deterministic when we exclude the wait states.

6 Experiments

We have performed experiments on RW-PC (a bounded-buffer protected by a reader-writer lock), BARBER (a solution to the sleeping barber problem)and AIRPORT (a specification for an airport ground traffic control simulation program). The properties verified are given in Table 2. The specifications of these examples along with the resulting Action Language files and the generated Java codes are available at *http://www.cs.ucsb.edu/~ bultan/tools/MV*. Here we give the comparison of two approaches for the RW-PC and AIRPORT instances. The first approach is using only JPF for verification. We provide a complete system with an actual controller class and client threads to JPF. The second approach is verifying the behavior specification using Action Language Verifier and verifying that the client threads behave according to the interface specification using JPF with stubs.

Table 1: Performance results for JPF with actual monitors

Problem Instance	Heuristic	TN-S	M	T	Heuristic	TN-S	M	T
RW-PC-1	DFS	2-2	21.61	20.06	BFS	2-2	17.72	50.78
RW-PC-1	DFS	2-3	33.90	30.57	BFS	2-3	32.54	90.71
RW-PC-1	DFS	2-4	44.80	43.81	BFS	2-4	42.01	145.16
RW-PC-1	DFS	2-5	49.17	59.65	BFS	2-5	55.46	218.44
RW-PC-1	DFS	2-6	73.07	76.69	BFS	2-6	67.59	305.25
RW-PC-1	DFS	2-7	84.54	99.20	BFS	2-7	89.84	417.96
RW-PC-1	DFS	3-2	$\uparrow$	$\uparrow$	BFS	3-2	$\uparrow$	$\uparrow$
AIRPORT-1	DFS	2	24.61	25.79	BFS	2	23.59	57.39
AIRPORT-1	DFS	3	329.70	1430.44	BFS	3	309.71	880.37
AIRPORT-1	DFS	4	$\uparrow$	$\uparrow$	BFS	4	$\uparrow$	$\uparrow$

Table 2: Performance results for interface-based verification (JPF with stubs)

Problem Instance	Property	ALV		JPF	
		VT	M1	TT	M2
RW-PC-1	$AG(nr > 0 \wedge count = x \Rightarrow AX(count = x))$	0.13	6.76	2.74	1.43
RW-PC-2	$AG(\neg busy \wedge count = x \Rightarrow AX(count \neq x \Rightarrow busy))$	0.08	6.78	2.74	1.43
RW-PC-3	$AG(EX(True))$	0.08	6.29	2.74	1.43
RW-PCP-1	$AG(nr > 0 \wedge count = x \Rightarrow AX(count = x))$	0.63	10.80	2.74	1.43
RW-PCP-2	$AG(\neg busy \wedge count = x \Rightarrow AX(count \neq x \Rightarrow busy))$	0.36	9.40	2.74	1.43
RW-PCP-3	$AG(EX(True))$	0.44	7.10	2.74	1.43
AIRPORT-1	$AG(numRW16R \leq 1 \wedge numRW16L \leq 1)$	0.35	23.09	3.33	2.15
AIRPORT-2	$AG(numC3 \leq 1)$	0.16	22.48	3.33	2.15
AIRPORTP-1	$AG(numRW16R \leq 1 \wedge numRW16L \leq 1)$	0.61	31.29	3.33	2.15
AIRPORTP-2	$AG(numC3 \leq 1)$	0.22	30.21	3.33	2.15
BARBER-1	$AG(chair \leq 1)$	0.62	7.24	1.43	0.77
BARBER-2	$AG(open \leq 1)$	0.66	7.24	1.43	0.77
BARBER-3	$AG(barber \leq 1)$	0.71	7.24	1.43	0.77
BARBERP-1	$AG(chair \leq 1)$	0.54	7.22	1.43	0.77
BARBERP-2	$AG(open \leq 1)$	0.55	7.22	1.43	0.77
BARBERP-3	$AG(barber \leq 1)$	0.54	7.22	1.43	0.77
PC	$AG(count \leq size)$	0.03	0.61	1.87	1.41
RW	$AG(busy \Rightarrow nr = 0)$	0.01	6.30	1.93	1.37
RWP	$AG(busy \Rightarrow nr = 0)$	0.42	6.94	1.93	1.37

We report the performance of JPF on verifying the whole system using both depth first search and breadth first search heuristics in Table 1. The column labelled TN-S shows the number of client threads and the buffer size (for the bounded-buffer example). The memory usage (in MBytes) is shown in the column labelled M, and the execution time (in seconds) is displayed in the column labelled T. For the RW-PC 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 1. JPF runs out of (512MB) memory (denoted by $\uparrow$) for buffer size 2 and 3 client threads in both heuristics. The performance of JPF drops dramatically when the number of client threads increases because of the increase in the number of possible interleavings. JPF runs out of 512MB memory for 4 client threads for the AIRPORT example.

Table 2 shows the performance of the second approach where the concurrency controller specification is verified using the Action Language Verifier and the thread behavior is verified using JPF with stubs. The columns labelled VT and M1 denote the time (in seconds) spent and the memory usage (in MBytes) for verification by the Action Language Verifier. The columns labelled TT and M2 denote the time spent (in seconds) and the memory usage (in MBytes) for the client thread verification using JPF and stubs. The problem instances marked with P are verified for arbitrary number of threads using counting abstraction. In the second approach for the RW-PC case, the local properties of RW and PC are verified separately. The global properties that refer to both RW and PC variables are verified in the composed controller RW-PC. The behavior is verified for any size of the buffer using Action Language Verifier. Action Language Verifier also enables us to verify the controller behavior for any number of client threads using counting abstraction. As for the verification of the client threads, the memory usage improves significantly since we use stubs which have finite reachable state spaces. JPF successfully verifies the problem instances with stubs without running out of memory.

7 Conclusions

Decoupling the verification of the concurrency controllers from the verification of the threads which use them has several advantages. It enables us to use infinite-state verification techniques to verify a controller with parameterized constants or arbitrary number of threads. We are also able to use finite state program verification tools on the client side which enables us to verify arbitrary Java threads without any restrictions. Using the interface specifications we can verify the client threads using stubs generated from the controller interfaces instead of the controllers themselves. Our experiments show that this approach improves the efficiency of thread verification significantly. Additionally, using our modular approach one can design complex concurrency controllers by composing interfaces of simpler concurrency controllers.

References

[BGP99] T. Bultan, R. Gerber, and W. Pugh. Model-checking concurrent systems with unbounded integer variables: Symbolic representations, approximations, and experimental results. *ACM Transactions on Programming Languages and Systems*, 21(4):747–789, July 1999.

[BHPV00] G. Brat, K. Havelund, S. Park, and W. Visser. Java pathfinder - a second generation of a Java model checker. In *Proceedings of the Workshop on Advances in Verification*, 2000.

[Bul00] T. Bultan. Action language: A specification language for model checking reactive systems. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, pages 335–344, June 2000.

[BYK01] 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.

[Car96] T. Cargill. Specific notification for Java thread synchronization. In *Proceedings of the 3rd Conference on Pattern Languages of Programs*, 1996.

[CdAHM02] A. Chakrabarti, L. de Alfaro, T.A. Henzinger, and F.Y.C. Mang. Synchronous and bidirectional component interfaces. In E. Brinksma and K.G. Larsen, editors, *Proceedings of the 14th International Conference on Computer Aided Verification (CAV 2002)*, Lecture Notes in Computer Science 2404, pages 414–427. Springer-Verlag, 2002.

[CH74] R. H. Campbell and A. N. Haberman. The specification of process synchronization by path expressions. *Lecture Notes in Computer Science, Operating Systems*, 16:89–102, 1974.

[dAH01a] L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE'01)*, pages 109–120, 2001.

[dAH01b] L. de Alfaro and T. A. Henzinger. Interface theories for component-based design. In *Proceedings of the First International Workshop on Embedded Software (EMSOFT '01)*, Lecture Notes in Computer Science 2211. Springer-Verlag, 2001.

[DDHM02] 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 (ICSE 2002)*, 2002.

[Del00] G. Delzanno. Automatic verification of parameterized cache coherence protocols. In *Proceedings of the 12th International Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 53–68. Springer-Verlag, 2000.

[Lea99] D. Lea. *Concurrent Programming in Java*. Addison-Wesley, Reading, Massachusetts, 1999.

[PDH99] 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*, 1999.

[YKB02] 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 (ISSTA 2002)*, pages 169–179, 2002.

[YKTB01] T. Yavuz-Kahveci, M. Tuncer, and T. Bultan. A library for composite symbolic representations. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, volume 2031 of *Lecture Notes in Computer Science*, pages 335–344. Springer-Verlag, April 2001.

[Zho97] C. Zhong. *Modeling of Airport Operations Using An Object-Oriented Approach.* PhD thesis, Virginia Polytechnic Institute and State University, 1997.