Specification, Verification, and Synthesis of Concurrency Control Components *

Tuba Yavuz-Kahveci Tevfik Bultan
Computer Science Department
University of California
Santa Barbara, CA 93106, USA
{tuba,bultan}@cs.ucsb.edu

ABSTRACT

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 processes. In this paper, we propose an approach in which 1) the concurrency control component of a concurrent program is formally specified, 2) it is verified automatically using model checking, and 3) the code for concurrency control component is automatically generated. We use monitors as the synchronization primitive to control access to a shared resource by multiple concurrent processes. Since our approach decouples concurrency control component from rest of the implementation it is scalable. We demonstrate its scalability by applying our approach to a case study on Airport Ground Traffic Control.

We use Action Language to specify the concurrency control component of a system. Action Language is a specification language for reactive software systems. It is supported by an infinite-state model checker that can verify systems with boolean, enumerated and unbounded integer variables. Our code generation tool automatically translates the verified Action Language specification into a Java monitor. We discuss correctness of our translation algorithm in terms of preservation of the verified properties. Our translation algorithm employs symbolic manipulation techniques and the specific notification pattern to generate an optimized monitor class by eliminating the context switch overhead introduced as a result of unnecessary thread notification. Using counting abstraction, we show that we can automatically verify the monitor specifications for arbitrary number of threads.

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

Keywords

concurrent programming, monitors, infinite-state model checking, specification languages

1. INTRODUCTION

Implementing a concurrent program is an error prone task. A concurrent programmer has to keep track of not only the possible values of the variables of the program, but also the states of its concurrent processes. Failing to use concurrency constructs such as semaphores or monitors correctly results in run-time errors such as deadlocks and violation of safety properties. Conventional validation techniques such as testing become ineffective for concurrent programs since the state space of a concurrent program increases exponentially both with the number of variables and the number of concurrent processes in it.

Monitors are a programming language construct introduced to ease the difficult task of concurrent programming [13]. Structured programming languages help programmers in keeping track of the variables by providing abstractions such as procedures and associated scoping rules to localize variable access. Monitors are a similar mechanism for structuring concurrent programs, they provide scoping rules for concurrency.

Since monitors are an integral part of Java, recently, concurrent programming using monitors gained increased attention [15]. A monitor consists of a set of variables shared among multiple processes and a set of associated procedures for accessing them. Shared variables of a monitor are not accessible outside of its procedures. At any time, only one process is allowed to be active in a monitor. Processes synchronize using specific operations which lets them *wait* (i.e., sleep) until they receive a *signal* from another process. Wait and signal operations are coordinated using condition variables. Even though monitors provide a better abstraction for concurrent programming compared to other constructs such as semaphores, they are still error prone. Coordinating the wait and signal operations on several condition variables among multiple processes can be very challenging even for the implementation of simple algorithms.

In this paper we propose a new approach for developing reliable concurrent programs. First aspect of our approach is to start with a specification of the concurrency control component of the program rather than its implementation. We use monitors as the underlying concurrency control primitive. We present a monitor model in Action Language [4]. Action Language is a specification language for reactive software systems. It supports both synchronous and asynchronous compositions and hierarchical specifications. We show that monitors can be specified in Action Language using asynchronously composed modules. Our monitor model in Action Language has one important aspect, it does not rely on condition variables. Semantics of Action Language lets us get rid of condition variables (with associated wait and signal operations) which simplifies the specification of a monitor significantly.

Most important component of our approach is the use of an automated verification tool for checking properties of monitor specifications. Action Language is supported by an infinite-state model checker that can verify or falsify (by producing counter-example behaviors) both invariance and liveness (or any CTL) properties of Action Language specifications [6]. In this paper we focus on verification of monitor invariants, however, the approach presented in this paper can be extended to full CTL. For the infinite-state systems that can be specified in Action Language, model checking is undecidable. Hence our verifier uses various heuristics to guarantee convergence. It does not produce false positives or negatives but its analysis can be inconclusive.

The last component of our approach is a code generation algorithm for synthesizing monitors from Action Language specifications. Our goal is not generating complete programs, rather, we are proposing a modular approach for generating concurrency control component of a program that manipulate shared resources.

We use a case study on Airport Ground Traffic Control [20] to show the effectiveness and scalability of our technique. This case study uses a fairly complex airport ground network similar to that of Seattle Tacoma International Airport. Our model checker can verify all the safety properties of the specification for this case study and our code generation tool automatically generates an optimized Java class in terms of the context switch overhead that would be incurred in a multithreaded application.

Recently, there has been several attempts in adopting model checking to verification of concurrent programs [8, 12]. These approaches translate a concurrent Java program to a finite model and then check it using available model checking tools. Hence, they rely on the ability of model checkers to cope with the state space explosion problem. However, to date, model checkers are not powerful enough to check implementations of concurrent programs. Hence, most of the recent work on verification of concurrent programs have been on efficient model construction from concurrent programs [8, 12, 11].

Our approach provides a different direction for creating reliable concurrent programs. It has several advantages: 1) It avoids the implementation details in the program which do not relate to the property to be verified. 2) There is no model construction problem since the specification language used has a model checker associated with it. 3) By pushing the verification to an earlier stage in software development (to specification phase rather than the implementation phase) it reduces the cost of fixing bugs. However, our approach is unlikely to scale to generation of complete programs. This would require the specification language to be more expressive and would introduce a model construction problem at the specification stage. Hence we focus on synthesizing concurrency control components which are correct by construction and can be integrated to a concurrent program safely. Another aspect of our approach which is different then the previous work is the fact that we are using an infinite state model checker rather then finite state techniques. Using our infinite state model checker we can verify properties of specifications with unbounded integer variables and arbitrary number of threads.

Rest of the paper is organized as follows. In Section 2 we describe our case study. In Section 3 we explain concurrency control with monitors and their implementation in Java. In Section 4 using our case study we discuss how monitors can be specified in Action Language. We also present a general template for specifying monitors in Action Language in that section. In Section 5 we discuss how monitor specifications can be automatically verified using Action Language Verifier. We also show that using counting abstraction monitor specifications can be verified for arbitrary number of threads. In Section 6 we present the algorithms for generating Java monitor classes from monitor specifications. We discuss correctness of the translation of an Action Language specification into a Java class in the same section. Finally, in Section 7, we state our conclusions and directions for future work.

2. AN AIRPORT GROUND TRAFFIC CONTROL CASE STUDY

We will present an Airport Ground Traffic Control case study to demonstrate the techniques proposed in this paper. Airport Ground Traffic Control handles allocation of airport ground network resources such as runways, taxiways, and gates to the arriving and departing airplanes. Airport Ground Traffic Control is safety critical. 51.5% of hull loss accidents from 1959 through 1996 were associated with airport ground operations such as taxi, takeoff, and landing [1]. Simulations play an important role for airport safety since they enable early prediction of possible runway incursions which is a growing problem at busy airports throughout the world. [20] presents a concurrent simulation program for modeling Airport Ground Traffic Control using Java threads. In this paper, we demonstrate that concurrency control component of such a program can be formally specified, automatically verified, and synthesized in our framework. We use the same airport ground network model used in [20] (shown in Figure 1) similar to Seattle/Tacoma International Airport. There are two runways: 16R and 16L. Runway 16R is used by arriving airplanes during landing. After landing, an arriving airplane takes one of the exits C3-C8. After taxiing on C3-C8, the arriving airplanes need to cross runway 16L. After crossing 16L, they continue on one of the taxiways B2, B7, B9-B11 and reach the gates in which they park. Departing airplanes use runway 16L for takeoff. The control logic for ground traffic of this airport must implement the following rules:

1. An airplane can land (takeoff) using 16R (16L) only if no airplane is using 16R (16L) at the moment.

2. An airplane taxiing on one of the exits C3-C8 can cross runway 16L only if no airplane is taking off at the moment.

3. An airplane can start using 16L for taking off only if none of the crossing exits C3-C8 is occupied at the moment. (Arriving airplanes have priority over departing airplanes.)

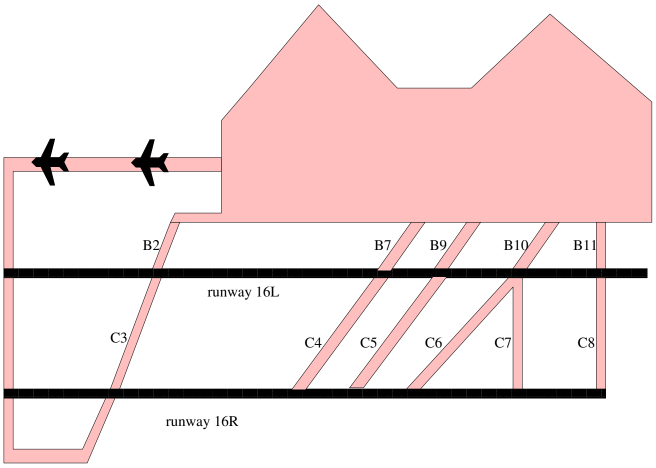

Figure 1: An airport ground network similar to that of the Seattle Tacoma International Airport

4. Only one airplane can use a taxiway at a time.

We give the Action Language specification of the Airport Ground Traffic Control system in Section 4. In Section 5 we discuss how we used the Action Language Verifier to automatically verify the properties of this system. In Section 6 we show the Java monitor class synthesized from the Action Language specification.

3. CONCURRENCY CONTROL WITH MONITORS

A monitor is a synchronization primitive that is used to control access to a shared resource by multiple concurrent processes. A monitor consists of a set of variables and procedures with the following rules: 1) The variables in a monitor can only be accessed through the procedures of the monitor. 2) No two process can execute procedures of the monitor at the same time. We can view the second rule as monitor having a mutual exclusion lock. Only the process that has the monitor lock can be active in the monitor. Any process that calls a monitor procedure has to acquire the monitor lock before executing the procedure and release it after it exits. This synchronization is provided implicitly by the monitor semantics, hence, the programmer does not have to explicitly write the acquire lock and release lock operations.

Monitors provide additional synchronization operations among processes based on condition variables. Two operations on condition variables are defined: *wait* and *signal*. A process that performs a wait operation on a condition variable sleeps and releases the monitor lock. It can only be awakened by a signal operation on the same condition variable. A waiting process that has been awakened has to re-acquire the monitor lock before it resumes operation. If there are no waiting processes, then signal operation is ignored (and forgotten, i.e., it does not affect processes which execute a wait later on). Wait and signal operations can be implemented using one wait queue per condition variable. When a process executes the wait operation on a condition variable it enters the corresponding wait queue. A signal operation on a condition variable removes one process from the corresponding wait queue and resumes its operation (after re-acquiring the monitor lock). In *signal and continue* semantics for the signal operation, signaling process keeps the monitor lock until it exits or waits. Different semantics and additional operations have also been used for signaling such as *signal and wait* semantics and *signalAll* operation [3].

Typically condition variables are used to execute a set of statements only after a guarding condition becomes true. To achieve this, a condition variable is created that corresponds to the guarding condition. The process which will execute the guarded statements tests the guarding condition and calls the wait on the corresponding condition variable if the guarding condition is false. Each process which executes a statement that can change the truth value of the guarding condition signals this to the processes that are waiting on the corresponding condition variable.

State of a monitor is represented by its variables. Set of states that are safe for a monitor can be expressed as a monitor invariant [3]. Monitor invariant is expected to hold when no process is accessing the monitor (i.e., it is not guaranteed to hold when a process is active within a monitor procedure).

3.1 Monitors in Java

Java is an object-oriented programming language that supports concurrent programming via threads and monitors. Each Java object has a mutual exclusion lock. A monitor in Java is implemented using the object locks and the `synchronized` keyword. A block of statements can be declared to be synchronized using the lock of an object `o` as `synchronized(o) { ... }`. This block can only be executed after the lock for the object `o` is acquired. Methods can also be declared to be `synchronized` which is equivalent to enclosing the method within a synchronized block using object `this`, i.e., `synchronized(this) { ... }`. A monitor object in Java is created by declaring a class with private variables that correspond to shared variables of the monitor. Then each monitor procedure is declared as a synchronized method to meet the mutual exclusion requirement.

Wait and signal operations are implemented as `wait` and `notify` methods in Java. However, in Java, each object has only one wait queue. This means that when there is a notify call, any waiting process in the monitor can wake up. If there is more than one condition that processes can be waiting for, awakened processes have to recheck the conditions they have been waiting for. Note that, if a process that was waiting for a different condition is awakened, then the notify call is lost. This can be prevented by using `notifyAll` method which wakes up all the waiting processes.

Using a single wait queue and `notifyAll` method one can safely implement monitors in Java. However, such an implementation will not be very efficient. To get better efficiency, one can use other objects (declared as members of the monitor class) as condition variables together with synchronized blocks on those objects. Since each object has a lock and an associated wait queue, this makes it possible to put processes waiting on different conditions to different queues. However, this implies that there will be more than one lock used in the monitor. (In addition to monitor lock there will be one lock per condition variable.) Use of multiple locks in Java monitor classes is prone to deadlocks and errors [15].

4. SPECIFICATION OF MONITORS

Although monitors provide a higher level of abstraction for concurrent programs compared to mechanisms such as semaphores, they can still be tedious and difficult to implement. We argue that Action Language can be used to specify monitors in a higher level of abstraction. Monitor specifications in Action Language do not rely on condition variables. Since in Action Language an action is executed only when its guard evaluates to true, we do not need conditional waits.

Figure 2 shows the Action Language specification of the Airport Ground Traffic Control case study. An Action Language specification consists of a set of module definitions. A module definition consists of variable declarations, a restrict expression, an initial expression, submodule definitions, action definitions, and a module expression. Semantically, each module corresponds to a transition system with a set of states, a set of initial states and a transition relation. Variable declarations define the set of states of the module.

In Figure 2 we implemented the shared resources of the Airport Ground Traffic Control, which are runways and taxiways, as integer variables. Variables `numRW16R` and `numC3` denote the number of airplanes on runway 16R and on taxi way C3, respectively. We use enumerated variables (local variable `pc` in submodule `Airplane`) to encode states of an airplane. An arriving airplane can be in one of the following states: `arFlow`, `touchDown`, `taxiX`, and `parked`, where `X` denotes the location of the airplane in the taxiways. Similarly, a departing airplane can be in one of the following states: `parked`, `takeOff`, and `depFlow`.

Action Language is a modular language. An Action Language specification can be defined in terms of a hierarchy of modules. In Figure 2 module `main` has a submodule `Airplane`. Submodule `Airplane` models both arriving and departing airplanes and corresponds to a process type (or thread class in Java). Submodule `Airplane` has one local boolean variable (`pc`) which is used to keep track of the states an airplane can be in. Note that, each instantiation of a module will create different instantiations of its local variables.

Set of states can be restricted using a restrict expression. In Figure 2, variables `numC3`, `numRW16R`, and `numRW16L` are restricted to be greater than or equal to 0. Initial expression defines the set of initial states of a module. For instance, in Figure 2 variables `numRW16R`, `numRW16L`, and `numC3` are initialized 0. Initial and restrict expressions of the submodules are conjoined with the initial and restrict expressions of the main module to obtain the overall initial condition and the restrict expression, respectively.

Behavior of a module (i.e., its transition relation) is defined using a module expression. A module expression (which starts with the name of the module) is written by combining its actions and submodules using asynchronous (denoted by `|`) and/or synchronous (denoted by `&`) composition operators. For instance, module `Airplane` is defined in terms of asynchronous composition of its actions `reqLand`, `exitRW3`, and so on. Module `main` (which defines the transition relation of the overall system) is defined in terms of asynchronous composition of multiple instantiations of its submodule `Airplane`. The specification in Figure 2 specifies a system with more than two airplane processes. In Figure 2 only asynchronous composition is used.

Each atomic action in Action Language defines a single execution step. In an action expression for an action a, primed (or range) variables, $\textsc{rvar}(a)$, denote the next-state values for the variables and unprimed (or domain) variables, $\textsc{dvar}(a)$, denote the current-state values. For instance, action `exitRW3` in module `Airplane` indicates that when an arriving airplane is in touchDown state (`pc = touchDown`) if taxiway C3 is available (`numC3=0`) then in the next state run way 16R will have one less airplanes (`numRW16R'=numRW16L-1`) and taxi way C3 will not be used by one more airplane (`numC3'=numC3+1`) and the airplane will be in state `pc' = taxiFr16LB2`. Note that an airplane taxiing on taxi way C3 crosses run way 16L on its route and continues on taxi way B2 (see Figure 1).

Asynchronous composition of two actions is defined as the disjunction of their transition relations. However, we also assume that an action preserves the values of the variables which are not modified by itself. Formally, we extend the action expression $\textsc{exp}(a_1)$, for action a_1, by conjoining it with a frame constraint $\textsc{exp}'(a_1) \equiv \textsc{exp}(a_1) \wedge \textsc{same}(\textsc{rvar}(a_2) - \textsc{rvar}(a_1))$ where $\textsc{same}(V)$ denotes $\wedge_{v_i \in V} v_i' = v_i$ and $-$ denotes set difference. Similarly, we extend the expression for a_2, $\textsc{exp}(a_2)$, to $\textsc{exp}'(a_2)$. Then, we define $\textsc{exp}(a_1 \mid a_2)$ as

$$\textsc{exp}(a_1 \mid a_2) \equiv \textsc{exp}'(a_1) \vee \textsc{exp}'(a_2)$$

Asynchronous composition of two modules is defined similarly.

In Figure 2 monitor invariants that we expect the system to satisfy is written using the `spec`, `invariant` and `next` keywords at the end of the `main` module. In Action Language keywords `invariant`, `eventually` and `next` are synonyms for CTL operators AG, AF, and AX, respectively.

The specification given in Figure 2 specifies a solution to the Airport Ground Traffic Control without specifying the details about the implementation of the monitor. It is a high level specification compared to a monitor implementation, in the sense that, it does not introduce condition variables and waiting and signaling operations which are error prone.

We give a general template for specifying monitors in Action Language in Figure 3. It consists of a main module m and a list of submodules $m_1, \ldots, m_n$. The variables of the main module (denoted $\textsc{var}(m)$) define the shared variables of the monitor specification. Currently, available variable types in Action Language are boolean, enumerated and integer. This restriction comes from the symbolic manipulation capabilities of the Action Language Verifier (which can be extended as we discuss in [14]). We also allow declaration of parameterized constants. For example, a declaration such as `parameterized integer size` would mean that `size` is an unspecified integer constant, i.e., when a specification with such a constant is verified it is verified for all possible values of `size`.

Each submodule m_i corresponds to a process type, i.e., each instantiation of a submodule corresponds to a process. Each submodule m_i has a set of local variables ($\textsc{var}(m_i)$) and atomic actions ($\textsc{act}(m_i)$). Note that, in a monitor specification our goal is to model only the behavior of a process that is relevant to the properties of the monitor. Therefore, local variables $\textsc{var}(m_i)$ of a submodule should only include the variables that are relevant to the correctness of the monitor. The transition relation of a submodule is defined as the asynchronous execution of its atomic actions.

To simplify the abstraction and code generation algorithms we will present in the following sections we restrict the form of action expressions as follows: Given an action $a \in \textsc{act}(m_i)$,

```
module main()
  integer numRW16R, numRW16L, numC3 ...;
  initial: numRW16R=0 and numRW16L=0 numC3=0 ...;
  restrict: numRW16R>=0 and numRW16L>=0 numC3>=0 ...;
  module Airplane()
     enumerated pc {arFlow, touchDown, parked, depFlow,
                  taxiTo16LC3, taxiTo16LC4, taxiTo16LC5,
                  taxiTo16LC6, taxiTo16LC7, taxiTo16LC8,
                  taxiFr16LB2, taxiFr16LB7, taxiFr16LB9,
                  taxiFr16LB10, taxiFr16LB11, takeOff};
     initial: pc=arFlow or pc=parked;
     reqLand: pc=arFlow and numRW16R=0 and pc'=touchDown
        and numRW16R'=numRW16R+1;
     exitRW3: pc=touchDown and numC3=0 and pc'=taxiTo16LC3
        and numRW16R'=numRW16R-1 and numC3'=numC3+1;
     crossRW3: pc=taxiTo16LC3 and numRW16L=0 and numB2A=0
        and pc'=taxiFr16LB2 and numC3'=numC3-1 and
        numB2A'=numB2A+1;
     reqTakeOff: pc=parked and numRW16L=0 and numC3=0
        and numC4=0 and numC5=0 and numC6=0 and
        numC7=0 and numC8=0 and and pc'=takeOff and
        numRW16L'=numRW16L+1;
     leave: pc=takeOff and pc'=depFlow
        and numRW16L'=numRW16L-1;
     changeStatus: pc=depFlow and pc'=arFlow;
     . . .
     Airplane: reqLand | exitRW3 | exitRW4 | exitRW5
        | exitRW6 | exitRW7 | exitRW8 | crossRW3
        | crossRW4 | crossRW5 | crossRW6 | crossRW7
        | crossRW8 |  park2 | park7 | park9 | park10
        | park11 | reqTakeOff | leave | changeStatus;
  endmodule
  main: Airplane() | Airplane() | ... ;
  // Property 1 (P1)
  spec: invariant(numRW16R<=1 and numRW16L<=1)
  // Property 2 (P2)
  spec: invariant(numC3<=1)
  // Property 3 (P3)
  spec: invariant((numRW16L=0 and
                 numC3+numC4+numC5+numC6+numC7+numC8>0)
                 => next(numRW16L=0))
endmodule
```

Figure 2: Action Language Specification of the Airport Ground Traffic Control Case Study

$\mathrm{EXP}(a)$ can be written as

$$\mathrm{EXP}(a) \equiv d_l(a) \ \wedge \ r_l(a) \ \wedge \ d_s(a) \ \wedge \ r_s(a)$$

where $d_l(a)$ is an expression on unprimed local variables of module m_i ($\mathrm{VAR}(m_i)$), $r_l(a)$ is an expression on primed and unprimed local variables of m_i, $d_s(a)$ is an expression on unprimed shared variables ($\mathrm{VAR}(m)$), and $r_s(a)$ is an expression on primed and unprimed shared variables. For example, for the action `exitRW3` in Figure 2

$d_l(a) \equiv$ `pc=touchDown`
$r_l(a) \equiv$ `pc'=taxiTo16LC3`
$d_s(a) \equiv$ `numC3=0`
$r_s(a) \equiv$ `numRW16R'=numRW16R-1 and numC3'=numC3+1`

In the template given in Figure 3, transition relation of the main module m is defined as the asynchronous composition of its submodules, which defines the behavior of the overall system.

5. VERIFICATION OF MONITOR SPECIFICATIONS

module $m()$
  integer i_1, i_2, ...;
  boolean b_1, b_2, ...;
  enumerated e_1, { val_1, val_2, ... };
  ...
  parameterized integer p_1, p_2, ...;
  restrict: *restrictCondition*;
  initial: *initialCondition*;
  module $m_1()$
    integer ...
    boolean ...
    enumerated ...
    restrict: ...
    initial: ....
    a_1: ...;
    a_2: ...;
    ...
    a_{n_1}: ...;
    m_1: a_1 | a_2 | ... | a_{n_1};
  endmodule
  module $m_2()$
    ...
  endmodule
  ...
  module $m_n()$
    ...
  endmodule
  m: $m_1()$ | $m_1()$ | ... | $m_2()$ | $m_2()$ ...
  spec: *monitorInvariant*
endmodule

Figure 3: A Monitor Template in Action Language

Action Language Verifier [6] consists of 1) a compiler that converts Action Language specifications to composite symbolic representations, and 2) an infinite-state symbolic model checker which verifies (or falsifies) CTL properties of Action Language specifications. Action Language compiler translates an Action Language specification to a transition system $T = (S, I, R)$ that consists of a *state space* S, a *set of initial states* $I \subseteq S$, and a *transition relation* $R \subseteq S \times S$. Unlike the common practices in model checking, S can be infinite and R may not be total (i.e., there maybe states $s \in S$ for which there does not exist a s' such that $(s, s') \in R$). For the infinite state systems that can be specified in Action Language, model checking is undecidable. Action Language Verifier uses several heuristics to achieve convergence such as approximations based on truncated fixpoint computations and widening, loop-closures and approximate reachability analysis. Since we allow non-total transition systems also some fixpoint computations have to be modified [6].

For the monitor model given in Figure 3, the state space S is obtained by taking the Cartesian product of the domains of the shared variables of the main module ($\mathrm{VAR}(m)$) and the domains of the local variables of each submodule ($\mathrm{VAR}(m_i)$) for each instantiation. The transition relation of R of the overall system is defined as

$$R \equiv \bigvee_{i,j,k} r_{ijk}$$

where r_{ijk} corresponds to the action expression of action a_k in instantiation j of module m_i. Action Language parser renames local variables of each submodule m_i for each instantiation j to obtain r_{ijk}'s. Also, as explained above, action expressions are automatically transformed by the Action Language parser by adding the frame constraints (unmodified variables preserve their value). Initial states of the system is defined as

$$I \equiv I_m \wedge \bigwedge_{i,j} I_{ij}$$

where I_m denotes the initial condition for main module m, and I_{ij} denotes the initial condition for instantiation j of module m_i.

Composite Symbolic Library [14] is the symbolic manipulator used by the Action Language Verifier. It combines different symbolic representations using the *composite model checking* approach [5]. Generally, model checking tools have been built using a single symbolic representation such as BDDs [16] or polyhedra [2]. A composite model checker combines different symbolic representations to improve both the efficiency and the expressiveness of model checking. Our current implementation of the Composite Symbolic Library uses two basic symbolic representations: BDDs (for boolean and enumerated variables) and polyhedral representation (for integers). Since Composite Symbolic Library uses an object-oriented design, Action Language Verifier is polymorphic. It can dynamically select symbolic representations provided by the Composite Symbolic Library based on the variable types in the input specification. For example, if there are no integer variables in the input specification Action Language Verifier becomes a BDD-based model checker.

To analyze a system using Composite Symbolic Library, one has to specify its initial condition, transition relation, and state space using a set of *composite formulas*. A composite formula is obtained by combining integer arithmetic formulas on integer variables with boolean variables using logical connectives. Enumerated variables are mapped to boolean variables by the Action Language parser. Since integer representation in the Composite Symbolic Library currently supports only Presburger arithmetic, we restrict arithmetic operators to $+$ and $-$. However, we allow multiplication with a constant and quantification.

A composite formula, p, is represented in disjunctive normal form as

$$p \equiv \bigvee_{i=1}^{n} \bigwedge_{t=1}^{T} p_{it}$$

where p_{it} denotes the formula of basic symbolic representation type t in the ith disjunct, and n and T denote the number of disjuncts and the number of basic symbolic representation types, respectively. Our Composite Symbolic Library implements methods such as intersection, union, complement, satisfiability check, subset test, which manipulate composite representations in the above form. These methods in turn call the related methods of basic symbolic representations.

Action Language Verifier iteratively computes the fixpoints for the temporal operators using the symbolic operations provided by the Composite Symbolic Library. Action Language Verifier uses techniques such as truncated fixpoints, widening and collapsing operators to compute approximations of the divergent fixpoint computations [6]. However, Action Language Verifier does not generate false negatives or false positives. It either verifies a property or generates a counter-example or reports that the analysis is inconclusive. This is achieved by using appropriate type of approximations for the fixpoints (lower or upper approximation) based on the temporal property and the type of the input query (which could be verify or falsify).

5.1 Counting Abstraction

In the Action Language template for monitor specifications (Figure 3), each submodule is instantiated a fixed number of times. This means that the specified system has a fixed number of processes. For example, the specification in Figure 2 describes a system with a specific number of airplane processes and hence, any verification result obtained for this specification is only guaranteed for a system with a specific number of airplane processes. In this section we will present a technique called *counting abstraction* [9] that makes it possible to verify the properties of a monitor model for arbitrary number of processes. The basic idea is to define an abstract transition system in which the local states of the processes are abstracted away but the number of processes in each local state is counted by introducing a new integer variable for each local state. For this abstraction technique to work we need local states of submodules to be finite. For example, if a submodule has a local variable that is an unbounded integer, we cannot use the counting abstraction. Note that, shared variables (i.e. $\text{VAR}(m)$) can still be unbounded since they are not abstracted away.

Consider the specification in Figure 2. Each `Airplane` process has 16 local states. Note that, in the general case (Figure 3), each local state corresponds to a valuation of all the local variables of a submodule, i.e., the set of local states of a submodule is the Cartesian product of the domains of the local variables of that submodule. Given a module m_i, let S_i be the set of all possible valuations of its local variables $\text{VAR}(m_i)$, then we call S_i the set of local states of m_i. In the counting abstraction, we introduce a nonnegative integer variable to represent each local state of each submodule. I.e., for each submodule m_i and for each local state $s \in S_i$ of module m_i we declare a nonnegative integer variable i_s. For example, for the specification in Figure 2, we introduce 16 integer variables for the `Airplane` submodule: `arFlowC` for state `pc=arFlow` and `depFlowC` for state `pc=depFlow`. These variables will represent the number of processes that are in the local state that corresponds to them. For example, if `arFlowC` is 2 in the abstract system, this will imply that there are 2 processes in the corresponding states of the concrete system where `pc=arFlow` holds. Note that, there could be more than one concrete state that corresponds to an abstract state.

Once we defined the mapping between the states of the concrete system and the abstract system, next thing to do is to define the abstract transition relation, i.e., to translate the actions of the original system to the actions of the abstract system. Consider the action:

```
reqLand: pc = arFlow and numRW16R=0 and
    pc' = touchDown and numRW16R' = numRW16R + 1;
```

To translate this action to an action on the abstract system we only have to change the part of the expression using the current and next state local variables (i.e., `pc=arFlow` and `pc'=touchDown`). The part of the expression on current and next state shared variables (i.e., `numRW16R=0` and `numRW16R'=numRW16R+1`) will remain the same. Since we restricted all local variables to be finite, without loss of generality, we can assume that all local variables are boolean variables (as we discussed in Section 5 Action Language Verifier translates enumerated variables to boolean variables). As we stated in Section 4 we assume that the action expression is in the form:

$$\text{EXP}(a) \equiv d_l(a) \ \wedge \ r_l(a) \ \wedge \ d_s(a) \ \wedge \ r_s(a)$$

where $d_l(a)$ is a boolean logic formula on local domain variables and $r_l(a)$ is a boolean logic formula on local domain variables. Since we are assuming that local variables can only be boolean, we do not need to have domain variables in $r_l(a)$. We can transform any action expression to a set of equivalent action expressions in this form by splitting disjuncts that involve both local and shared variables. For the action `reqLand` d_l is `pc=arFlow` and r_l is `pc'=touchDown`.

Let $S_d \subseteq S_i$ be the set of local states of the module m_i which satisfy expression d_l. We translate d_l to an expression on the variables of the abstract transition system by generating the expression

$$d_l^a \equiv \sum_{s \in S_d} i_s > 0$$

where i_s is the integer variable that represents the local state s. Note that, d_l^a indicates that there exists a process which is in a local state that satisfies d_l. For `reqLand` the formula we obtain is simply `arFlowC>0`, i.e., there exists a process in the local state `pc=arFlow`.

Let $S_r \subseteq S_i$ be the set of local states of the module i which satisfy expression r_l. We translate r_l to an expression on the variables of the abstract transition system by generating the expression

$$\begin{aligned} r_l^a \equiv \ & \textstyle\bigvee_{s \in S_d, t \in S_r, s \neq t} (i'_t = i_t + 1 \ \wedge \ i'_s = i_s - 1 \\ & \qquad \wedge \ \textstyle\bigwedge_{p \neq s, p \neq t} i'_p = i_p) \\ & \vee \ \textstyle\bigvee_{s \in (S_d \cap S_r)} (i'_s = i_s \ \wedge \ \textstyle\bigwedge_{p \neq s} i'_p = i_p) \end{aligned}$$

The first disjunction enumerates all possible local current and next state pairs for the action and updates the counters accordingly. The second disjunct takes into account the cases where the local state of the process does not change. For the action `reqLand` we obtain the following expression: `touchDownC'=touchDownC+1 and arFlowC'=arFlowC'-1`. Then, the abstraction of action `reqLand` is:

```
reqLand: arFlowC>0 and numRW16R=0 and
   numRW16R'=numRW16R+1 and arFlowC'=arFlowC-1
   and touchDownC'=touchDownC+1;
```

After generating the abstract state-space and the abstract transition relation, last component of the abstraction is to translate the initial states. First, for each submodule m_i in Figure 3 we declare a nonnegative parameterized integer constant num_m_i which denotes the number of instances of module m_i. By declaring this constant parameterized we guarantee that the verified properties will hold for all possible number of instantiations of each submodule. Let $init_i$ denote the local initial expression of a submodule m_i and let S_{init} denote the set of local states of the module m_i which satisfy expression $init_i$. Then we add the following constraint to the initial expression of the abstract transition system:

$$init_i^a \equiv \sum_{s \in S_{init}} i_s = num\_m_i \ \wedge \ \bigwedge_{s \notin S_{init}} i_s = 0$$

for each submodule m_i in m **do**
  Declare a parameterized integer num_m_i in m
  Add $num\_m_i \geq 0$ to the restrict expression of m
  Remove local variable declarations of module m_i
  for each local state $s \in S_i$ of module m_i **do**
    Declare an integer variable i_s in m_i
    Add $i_s \geq 0$ to the restrict expression of m_i
  Replace initial expression $init_i$ of module m_i with
    $\sum_{s \in S_{init_i}} i_s = num\_m_i \ \wedge \ \bigwedge_{s \notin S_{init_i}} i_s = 0$
      where S_{init_i} are the set of local states
      of m_i which satisfy $init_i$
  for each action a in module m_i **do**
    Replace $d_l(a)$ with
      $\sum_{s \in S_d} i_s > 0$
        where S_d is the set of local states
        of m_i that satisfy $d_l(a)$
    Replace $r_l(a)$ with
      $\bigvee_{s \in S_d, t \in S_r, s \neq t} (i'_t = i_t + 1 \ \wedge \ i'_s = i_s - 1$
      $\wedge \ \bigwedge_{p \neq s, p \neq t} i'_p = i_p)$
      $\vee \ \bigvee_{s \in (S_d \cap S_r)} (i'_s = i_s \ \wedge \ \bigwedge_{p \neq s} i'_p = i_p)$
        where S_r is the set of local states
        of m_i that satisfy $r_l(a)$

Figure 4: Algorithm for counting abstraction

For the specification in Figure 2 we create a nonnegative parameterized integer constant `numAirplane`. Using the initial condition of the submodule `Airplane`, which is `pc=arFlow or pc=depFlow`, we obtain the following constraint

```
arFlowC + parkedC=numAirplane and touchDownC=0 and
taxiTo16LC3C=0 and taxiTo16LC4C=0 and taxiTo16LC5C=0
and taxiTo16LC6C=0 and taxiTo16LC7C=0 and taxiTo16LC8C=0
and taxiFr16LB2C=0 and taxiFr16LB7C=0 and taxiFr16LB9C=0
and taxiFr16LB10C=0 and taxiFr16LB11C=0 and takeOffC=0
and depFlowC=0
```

and replace the initial constraint of `Airplane` submodule with this new constraint.

One can show that the monitor invariants verified on the abstract specification generated by the counting abstraction are also satisfied by the original monitor specification for arbitrary number of processes. We will not give a formal proof here, however, such a proof can be constructed by defining an abstraction function between the state space of the original specification and the state space of the abstract specification generated by the counting abstraction [10].

5.2 Experimental Results

Table 5.2 shows the performance of the Action Language Verifier for the Airport Ground Traffic Control monitor specification given in Figure 2. In the first column we denote the total number of processes in the specification. For example, the results from the first row are for the specification in Figure 2 with 2 `Airplane` processes. CT denotes the time spent in constructing the composite symbolic representations for the transition relation and the initial states of the the input specification (including the parsing time). VT denotes the verification time for each property. Although the input is an infinite state system (since `numC3, numRW16R, and numRW16L` are unbounded variables) the verification time scales very well. This is due to the efficiency of the composite symbolic representation and the BDDs. If we had partitioned the transition system to eliminate the boolean variables (as is done in most infinite state model checkers) we would have obtained 2^{64} partition classes for

Table 1: Verification Results For Airport Ground Traffic Control Specification. xA (xD) denotes x many arriving (departing) airplane processes and x=P denotes arbitrary number of airplane processes. P1, P2, and P3 are the properties given in Figure 2. CT and VT denote transition system construction time and property verification time (in seconds), respectively.

Processes	CT	VT		
		P1	P2	P3
2	0.81	0.42	0.28	0.69
4	1.50	0.78	0.50	1.13
8	3.03	1.53	0.99	2.22
16	6.86	3.02	2.03	5.07
2A,PD	1.02	0.64	0.43	0.83
4A,PD	1.94	1.19	0.81	1.39
8A,PD	3.95	2.28	1.54	2.59
16A,PD	8.74	4.6	3.15	5.35
PA,2D	1.67	1.31	0.88	3.94
PA,4D	3.15	2.42	1.71	5.09
PA,8D	6.40	4.64	3.32	7.35
PA,16D	13.66	9.21	7.02	12.01
PA,PD	2.65	0.99	0.57	0.43

the fourth instance in Table 5.2. Mapping the boolean variables to integers on the other hand would have created 64 more integer variables, increasing the cost of arithmetic constraint manipulation (which is not likely to scale as well as BDDs).

We used counting abstraction to verify the Airport Ground Traffic Control monitor specification for arbitrary number of arriving and departing airplanes. First we verified specifications for a fixed number of arriving airplanes and an arbitrary number of departing airplanes by using counting abstraction only on the states of departing airplanes. For example, the row 4A,PD in Table 5.2, denotes the case with 4 arriving airplanes and an arbitrary number of departing airplanes. Although counting abstraction generates an integer variable for each local state of an airplane process, the experimental results in Table 5.2 shows that it scales well. In fact, the case where both states of the arriving and the departing airplanes are abstracted (PA,PD) properties are verified faster compared to some other cases. This is possibly due to the fact that counting abstraction, in a way, simplifies the system by abstracting away the information about individual processes. For example, in the abstract transition system it is not possible to determine which airplane is in which state, we can only keep track of the number of airplanes in a particular state.

We verified a large number of concurrent system specifications using Action Language Verifier including other monitor specifications such as monitors for sleeping barber problem, readers-writers problem and bounded buffers. Our experimental results are reported in [19].

6. SYNTHESIS OF MONITORS

In the monitor specification given in Figure 2, the shared variables such as `numRW16R` and `numC3` represent the resources that will be shared among multiple processes. Submodule `Airplane` specify the type of processes that will share these resources. Our goal is to generate a monitor class in Java from monitor specifications such as the one given in Figure 2. First, we will declare the shared variables of the monitor specification (for example, `numRW16R` and `numC3` in Figure 2) as private fields of the monitor class. Hence, these variables will only be accessible to the methods of the monitor class.

We will not try to automatically generate code for the threads that will use the monitor. This would go against the modularization principle provided by the monitors. Rather, we will leave the assumption that the threads behave according to their specification as a proof obligation. In general, a submodule in a monitor specification (Figure 3) should specify the most general behavior of the corresponding thread, or, equivalently, it should specify the minimum requirements for the corresponding thread for the monitor to execute correctly. Since the specifications about the local behavior of the threads are generally straightforward (such as an `Airplane` process should not execute `exitRW3` action before executing `reqLand`) we think that it would not be too difficult for the concurrent programmer to take the responsibility for meeting these specifications.

We will generate a monitor method corresponding to each action of each submodule in the monitor specification. Consider the action:

```
reqLand: pc=arFlow and numRW16R=0 and
   numRW16R' = numRW16R + 1 and pc'=touchDown;
```

We are not interested in the expressions on local variable `pc` of the submodule `Airplane`. As we discussed above, we are only generating code for the monitor class which only has access to the shared variables. For the action `exitRW3` removing the expressions on the local variables leaves us with the expression

```
reqLand: numRW16R=0 and numRW16R'=numRW16R + 1;
```

To implement this action as a monitor method we first have to check the guarding condition `numRW16R=0` and then update `numRW16R`. However, if the guarding condition does not hold, we should wait until a process signals that the condition might have changed. A straightforward translation of this action to a monitor method would be

```
public synchronized void reqLand() {
  while (!(numRW16R==0))
    wait();
  numRW16R=numRW16R+1;
  notifyAll();
}
```

The reason we call the `notifyAll` method at the end is to wakeup processes that might be waiting on a condition related to variable `numRW16R` which has just been updated by this action. Also note that the `wait` method is inside a while loop to make sure that the guard still holds after the thread wakes up. In the above translation, we used `synchronized` keyword to establish atomicity. Note that atomicity in Java is established only with respect to other methods or blocks which are also synchronized. So for this approach to work we have to make sure that shared variables are not modified by any part of the program which is not synchronized. We can establish this by declaring shared variables as private variables in the monitor class and making sure that all the methods of the monitor class are synchronized.

Using this straightforward approach, we can translate a monitor specification (based on the template given in Figure 3) to a Java monitor class using the following rules: 1) Create a monitor class with a private variable for each shared variable of the specification. 2) For each action in each submodule, create a synchronized method in the monitor class. 3) In the method for action a start with a while loop which checks if $d_s(a)$ is true and waits if it is not. Then, put a set of assignments to update the variables according to the constraint in $r_s(a)$. After the assignments, call `notifyAll` method and exit. We will call this translation *single-lock implementation* of the monitor since it uses only *this* lock of the monitor class.

6.1 Specific Notification Pattern

The *single-lock implementation* described above is correct but it is inefficient [7, 17]. If we implement the Airport Ground Traffic Control monitor using the above scheme a `reqLand` action would awaken all the airplane threads that are sleeping. However, departing airplane threads should be awakened only when status of runway 16L or one of the taxi ways in C3-C8 changes (when variables `numRW16L`, `numC3`, `numC4`, `numC5`, `numC6`, `numC7`, and `numC8` are updated) and they do not need to be awakened on an update to status of runway 16R (when `numRW16R` is updated). Using different condition variables for each guarding condition improves the performance by awakening only related threads and eliminating the overhead incurred by context switch for threads which do not need to be awakened. In [7] using separate objects to wait and signal for separate conditions is described as a Java design pattern called *specific notification pattern.*

Figure 5 illustrates implementation of Airport Ground Traffic Control monitor in Java using specific notification pattern. The method for action `reqLand` calls `Guard_reqLand` method in a while loop till it returns true. If it returns false it waits on the condition variable `reqLandCond`. Any action that can change the guard for `reqLand` from false to true notifies the threads that are waiting on condition variable `reqLandCond` using `reqLandCond`. If the guard `(numRW16R==0)` is true then `Guard_reqLand` method increments the number of airplanes using runway 16R (`numRW16R=numRW16R+1`) atomically and returns true. Since executing `reqLand` can only change action `reqLand`'s guard from false to true, only threads that are waiting on condition variable `reqLandCond` are notified before method `reqLand` returns.

Action `leave` does not have a guard, i.e., its execution does not depend on the state of shared variables of the monitor. Hence, the method for action `leave` does not need to wait to decrement the number of airplanes on runway 16L (`numRW16L=numRW16L-1`). After updating `numRW16L` however it notifies the threads waiting on the condition variables `crossRW3Cond` and `reqTakeOffCond` since executing the assignments in method `leave` can change the guard of actions `crossRW3` and `reqTakeOff` from false to true.

We will give an algorithm for generating Java code from monitor specifications in Action Language using specific notification pattern below. As stated before, we will assume that each action expression is in the form:

$$\text{EXP}(a) \equiv d_l(a) \ \wedge \ r_l(a) \ \wedge \ d_s(a) \ \wedge \ r_s(a)$$

where $d_l(a)$ is an expression on unprimed local variables of module m_i ($\text{VAR}(m_i)$), $r_l(a)$ is an expression on primed and unprimed local variables of m_i, $d_s(a)$ is an expression on unprimed shared variables ($\text{VAR}(m)$), and $r_s(a)$ is an expression on primed and unprimed shared variables. Since we are not interested in the local states of the processes we will only use $d_s(a)$ and $r_s(a)$ in the code generation for the monitor

```
public class AirPortGroundTrafficControl{
  private int numC3;
  private int numRW16L, numRW16R;
  . . .
  private Object exitRW3Cond;
  private Object reqTakeOffCond;
  private Object reqLandCond;
  private Object crossRW3Cond;
  . . .
  public AirPortGroundTrafficControl() {
    numC3=0;
    numRW16L=0;
    numRW16R=0;
    . . .
    exitRW3Cond=new Object();
    reqTakeOffCond=new Object();
    reqLandCond=new Object();
    crossRW3Cond=new Object();
    . . .
  }
  private synchronized boolean Guard_reqLand() {
    if (numRW16R==0) {
      numRW16R=numRW16R+1;
      return true;
    }
    else return false;
  }
  public void reqLand() {
    synchronized(reqLandCond) {
      while(!Guard_reqLand())
        try{
          reqLandCond.wait();
        }
        catch(InterruptedException e) {
        }
    }
    synchronized(reqLandCond) {reqLandCond.notify();}
  }
  private synchronized boolean Guard_exitRW3() {
    if (numC3==0) {
      numC3=numC3+1;
      numRW16R=numRW16R-1;
      return true;
    }
    else return false;
  }
  public void exitRW3() {
    synchronized(exitRW3Cond) {
      while(!Guard_exitRW3())
        try{
          exitRW3Cond.wait();
        }
        catch(InterruptedException e) {
        }
    }
    synchronized(reqTakeOffCond) {reqTakeOffCond.notify();}
    synchronized(reqLandCond) {reqLandCond.notify();}
    // other notifications
    . . .
  }
  public void leave() {
    synchronized(this) {
      numRW16L=numRW16L-1;
    }
    synchronized(crossRW3Cond) {crossRW3Cond.notify();}
    synchronized(reqTakeOffCond) {reqTakeOffCond.notify();}
    // other notifications
    . . .
  }
  . . .
}
```

Figure 5: AirportGroundTrafficControl Class Using specific notification pattern

```
for each action a do
    if d_s(a) ≠ true then
        mark a as guarded
        create condition variable cond_a
    else
        mark a as unguarded
    for each action b s.t. a ≠ b do
        if POST(¬d_s(b), EXP(a)) ∩ d_s(b) ≠ ∅ then
            add cond_b to notification list of a
```

Figure 6: Extracting Synchronization Information

methods. Let $guard_a$ denote a Java expression equivalent to $d_s(a)$. We will also assume that $r_s(a)$ can be written in the form $r_s(a) \equiv \bigwedge_{v \in \text{RVAR}(a)} v' = e_v$ where e_v is an expression on domain variables in $\text{VAR}(m)$. Let $assign_a$ denote a set of assignments in Java which correspond to $r_s(a)$.

To use the specific notification pattern in translating Action Language monitor specifications to Java monitors we need to associate the guard of each action with a lock specific to that action. Let a be an action with a guard, $guard_a$, and let $cond_a$ be the condition variable associated with a. The thread that calls the method that corresponds to action a will wait on $cond_a$ when $guard_a$ evaluates to false. Any thread that calls a method that corresponds to another action, b, that can change the truth value of $guard_a$ from false to true will notify $cond_a$. Hence, after an action execution, only the threads that are relevant to the updates performed by that action will be awakened.

The algorithm given in Figure 6 generates the information about the synchronization dependencies among different actions needed in the implementation of the specific notification pattern. For each action a in each submodule m_i the algorithm decides whether action a is *guarded* or *unguarded* by checking the expression $d_s(a)$. If $d_s(a)$ is true (meaning that there is no guard) then the action is marked as unguarded. Otherwise, it is marked as guarded and a condition variable, $cond_a$, is created for action a. Execution of an unguarded action does not depend on the shared variables, hence, it does not need to wait on any condition variable. Next, the algorithm finds all the actions that should be notified after action a is executed. We can determine this information by checking for each action $b \neq a$, if executing action a when $d_s(b)$ is false can result in a state where $d_s(b)$ is true. If this is possible, then the condition variable created for action b, $cond_b$, is added to the notification list of action a, which holds the condition variables that must be notified after action a is executed.

Figure 7 shows translation of guarded and unguarded actions to Java [17]. For each guarded action a a specific notification lock, $cond_a$ is declared and one private method and one public method is generated. The private method $Guarded\_Execute_a$ is synchronized on *this* object. If the guard of action a is true then this method executes assignments in $assign_a$ and returns true. Otherwise, it returns false. Method $Guarded\_Wait_a$ first gets the lock for $cond_a$. Then it runs a while loop till $Guarded\_Execute_a$ method returns true. In the body of the while loop it waits on $cond_a$ till it is notified by some thread that performs an update that can change truth value of method $guard_a$ and, therefore, $Guarded\_Execute_a$. For each unguarded action a a single public method $Execute_a$ is produced. This method

```
private Object cond_a = new Object();
public void Guarded_Wait_a() {
    synchronized(cond_a) {
        while (!Guarded_Execute_a()) {
            try {
                cond_a.wait();
            }
            catch(InterruptedException e) {
            }
        }
    }
}

private synchronized boolean Guarded_Execute_a() {
    if (guard_a) {
        assign_a;
        return true;
    } else return false;
}
```

(a)

```
public void Execute_a() {
    synchronized(this) {
        assign_a;
    }
}
```

(b)

Figure 7: Translation of (a) guarded and (b) unguarded actions

first acquires the lock for `this` object. Then executes the assignments $assign_a$ of the corresponding action. Before exiting the public methods $Guarded\_Wait_a$ and $Execute_a$, synchronized($cond_b$) { $cond_b$.notifyAll(); } is executed for each action b in the notification list of action a (note that this is not shown in Figure 7).

6.2 Correctness of Monitor Synthesis

We discuss correctness of translating a monitor specification in Action Language to a monitor class in Java in terms of the preservation of the verified properties by the generated Java monitor class. The correctness of the translation can be divided to two steps 1) Showing that the verified properties are preserved by the *single-lock implementation* of the Action Language specification. 2) Showing the equivalence between the single-lock implementation and the specific notification pattern implementation. The proof of correctness of specific notification pattern (step 2) is given in [17]. The algorithm given in Figure 6 extracts the necessary information in order to generate a Java monitor class that correctly implements the specific notification pattern. Below we discuss that the *single-lock implementation* of an Action Language monitor specification preserves the verified properties (step 1).

Below, we will give a set of assumptions under which the monitor invariants that are verified on an Action Language specification of a monitor are preserved by its single-lock implementation as a Java monitor class.

1. *Initial Condition:* The set of program states immediately after the constructors of the monitor and the threads are executed satisfy the initial expression of

the Action Language specification.

2. *Atomicity:* The observable states of the monitor are defined as the program states where *this* lock of the monitor is available, i.e., the states where no thread is active in the monitor.

3. *Thread Behavior:* The local behavior of the threads are correct with respect to the monitor specification.

4. *Scheduling:* If there exists an enabled action then an enabled action will be executed.

Assuming that the above conditions hold we claim that the single-lock implementation of the Action Language monitor specification satisfies the monitor invariants verified by the Action Language Verifier.

7. CONCLUSIONS AND FUTURE WORK

We think that our approach of combining specification, verification and synthesis, presented in this paper, can provide the right cost-benefit ratio for adaptation of automated verification techniques in practice. Writing a monitor specification has three major benefits: 1) It is a higher-level specification of a solution then a monitor implementation since it eliminates the need for condition variables and wait and signal operations. 2) Action Language specifications can be verified with Action Language Verifier. 3) Verified monitor specifications in Action Language can be automatically translated into Java monitor implementations where the correctness of the implementation is guaranteed by construction.

We are working on the integration of the automated code generation and automated counting abstraction algorithms to the Action Language Verifier. We think that our approach is applicable to interesting, real-world applications as demonstrated by our case study. For our approach to be applicable to a wider range of systems, we would like to extend our techniques to systems with boolean or integer arrays and recursive data structures (such as linked lists). We are working on both of these problems. We think that we can provide some analysis for arrays using uninterpreted functions. For analyzing specifications with recursive data structures, we are currently integrating the shape analysis technique [18] to Composite Symbolic Library.

Our verification tools Composite Symbolic Library and Action Language Verifier are available at:
http://www.cs.ucsb.edu/~bultan/composite/

8. REFERENCES

[1] Statistical summary of commercial jet aircraft accidents, worldwide operations 1959-1995, 1996.

[2] R. Alur, T. A. Henzinger, and P. Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22(3):181–201, March 1996.

[3] G. R. Andrews. *Concurrent programming, Principles and Practice.* Benjamin/Cummings Publishing Co., 1991.

[4] 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.

[5] T. Bultan, R. Gerber., and C. League. Composite model checking: Verification with type-specific symbolic representations. *ACM Transactions of Software Engineering and Methodology*, 9(1):3–50, January 2000.

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

[7] T. Cargill. Specific notification for java thread synchronization. In *International Conference on Pattern Languages of Programming*, 1996.

[8] 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. (ICSE)*, 2000.

[9] 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, 2000.

[10] G. Delzanno and T. Bultan. Constraint-based verification of client-server protocols. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, 2001.

[11] C. Demartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent java programs. *Software-Practice and Experience*, 29(7):577–603, 1999.

[12] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2000.

[13] C. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[14] T. Y. Kahveci, M. Tuncer, and T. Bultan. A library for composite symbolic representation. In *Proceedings of the Seventh International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, 2001.

[15] D. Lea. *Concurrent Programming in Java, Design Principles and Java.* Sun Microsystems, 1999.

[16] K. L. McMillan. *Symbolic model checking.* Kluwer Academic Publishers, Massachusetts, 1993.

[17] M. Mizuno. A structured approach for developing concurrent programs in java. *Information Processing Letters*, 1999.

[18] R. Wilhelm, M. Sagiv, and T. Reps. Shape analysis. In *Proceedings of the 9th International Conference on Compiler Construction*, volume 1781 of *Lecture Notes in Computer Science*, pages 1–17, 2000.

[19] T. Yavuz-Kahveci and T. Bultan. Heuristics for efficient manipulation of composite constraints. In *Proceedings of the 4th International Workshop on Frontiers of Combining Systems*, 2002.

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