Abstraction For Analytic Verification of Concurrent Software Systems

Michael Lowry[1] and **M. Subramaniam**[2]

[1] Computational Sciences Division, Code IC, MS 269-2 NASA Ames Research Centre Mofett Field, CA 94035
lowry@ptolemy.arc.nasa.gov

[2] Recom Technologies, Computational Sciences Division, Code IC, NASA Ames Research Center, Mofett Field, CA
subu@ptolemy.arc.nasa.gov

Abstract. This paper describes methods towards automatically generating abstract models of complex software systems. The abstraction methods take as input a software system and a property to be verified, and produce an abstracted model. The abstract models generate d can be automatically analyzed by current model checkers. Towards this goal we describe two techniques based on Dijkstra's weakest precondition calculus for performing control and data abstractions of software applications. The first of these, called *semantic slicing,* performs control abstractions by replacing portions of a program that do not affect the truth value of a given property by *idle* statements. In other words, semantic slicing maximally abstracts a program with respect to the property being verified. The result of semantic slicing is a highly reduced program that is behaviorally equivalent to th e original program with respect to the given property. The second technique performs data abstractions by using weakest preconditions. A finite state abstract model is automatically generated from a given program and a finite set of user-supplied *control predicates.* The abstraction technique preserves counterexamples for invariant (universal) properties. In order to deal with liveness (existential) properties we also discuss an approach based on explicit abstraction mappings over the domains of the program variables for generating an abstract model. No information is lost in abstraction in this technique if the mappings are a congruence with respect to the program operations. The effectiveness of these techniques are illustrated by several examples including a nontrivial example of a spacecraft controller being developed at NASA.

1 Motivation

Traditional testing approaches to debugging and V&V (verification and validation) have reached the limits of system complexity for which they can provide high assurance. As system complexity increases, both in terms of system behavior and in terms of internal structure, the number of test cases required to cover the range of possibilities and to cover the internal computational paths rises exponentially. In particular, concurrent real-time systems such as spacecraft controllers are notoriously difficult to debug and verify by traditional V&V approaches. This is often due to subtle interactions between components and the fact that the system's behavior may depend upon timing. For example, Mars pathfinder repeatedly reset itself, thereby losing days of potential science data, due to a subtle priority inversion bug. Another conccurrency bug led to the first Brazilian microspacecraft being inoperable for six months until a formal methods team found the error. A bug in such a system may manifest itself only intermittently during testing, making it difficult to pinpoint the location of the bug in the software. In fact, a timing bug may never manifest itself until actual flight conditions, at which point an error condition may arise that causes the failure of the entire mission. Multiple levels of redundancy can mitigate the effects of some errors, but it is clear that catastrophic failures can arise from unexpected conditions. Sometimes failure recovery systems are part of the problem, as were the manner in which watchdog timers interacted with other components on Mars pathfinder and the Brazilian microspacecraft.

In contrast, automated reasoning approaches to debugging and V&V are more comprehensive than traditional testing in that they replace individual test cases with symbolic calculation that covers whole swaths of the test space at once. Therefore, analytic V&V frequently uncovers serious design flaws that survive extensive traditional testing. An example is the Pentium floating point bug, which went undetected through a period of extensive testing prior to the initial release of the Pentium. After this costly error, the digital hardware industry invested heavily in analytic approaches to debugging. Analytic approaches for verification and debugging such as model checking have recently proven very successful in uncovering subtle design and implementation flaws in complex state-of-the-art hardware microprocessors and concurrency protocols.

Our main objective is to apply analytic verification techniques such as model checking to the complex software systems being developed by NASA Space and Aeronautic Enterprises to provide high assurance software systems. Our primary focus is on complex, concurrent/distributed and real-time software systems. These include: autonomy software, spacecraft and rover task programs that operate in concurrent and distributed environments, software for managing distributed control of space missions, and software for distributed design of spacecraft and missions, real-time software for civilian avionics, etc.

An important issue in applying analytic verification techniques to such large scale systems is in generating an abstract mathematical model that can be automatically analyzed against formally stated requirements and designs. Currently, the abstract models have to be manually formulated by formal methods experts while carefully studying the application domain. This imposes considerable burden both in terms of time and cost on the use of analytic approaches. This is the major bottleneck preventing their wide-spread application. Developing abstract models for software systems is even more involved than that for the hardware applications, since the currently available model checkers are not well-suited for verifying software applications. For example, many model checkers do not support adequate language constructs in terms of data, control and concurrency. More importantly, most of the model checkers operate only on bounded models, which is an inherent limitation for software systems.

2 Outline of Approach

The main goal of this paper is to describe techniques that allow automatic generation of abstract models from software programs. The abstract models generated can be automatically analyzed by the current model checkers. Our approach towards this goal is two-fold: devise *translation* techniques from high level programming languages to input languages of model checking tools; devise *abstraction* techniques for abstracting data, control and concurrency aspects of a given program to generate an abstract model that is finite-state and is computationally tractable for analyses by model checking algorithms. This paper is primarily concerned with the latter aspect of developing abstraction techniques to generate a finite-state tractable model. For details on the language translation aspects of abstract model generation, the reader may refer to [13].

Our key idea in automatic generation of an abstract model is to use the property that needs to be verified of the given program as the basis for abstraction. Informally, control abstractions discard irrelevant portions of the program that do not affect the property. For data abstractions, the property along with a finite set of user supplied *control predicates* is used. The potentially unbounded set of values of the various data types in the program can then be reformulated in terms of their effect on these control predicates. This leads to a finite state abstract model since the number of control predicates is finite.

Weakest pre-conditions of a program statement with respect to a property or a set of control predicates can be used to determine both whether the statement affects the property and also to determine how the specified control predicates valuations are changed by executing the statement. This provides a basis for deriving an abstract interpretation of the program. The abstract model obtained via control abstractions is *exact* and preserves the truth value of the property, i.e. the property holds of the abstract model if and only if it holds of the program, since the constructs that do not affect the property are only discarded. The abstract model obtained via data abstractions is *conservative* in the sense that it preserves only the counterexamples of the property. So the abstract model could lead to "false negatives". This means that an abstract counterexample of a property needs to be checked against the concrete program to determine if it is a true counterexample.

A further limitation of the data abstraction approach based on weakest pre-conditions is that it is conservative only for invariant (universal) properties. For liveness (existential) properties, it appears infeasible to map their truth values in the abstract model generated by this approach to those in the program. In order to address this shortcoming, we describe an alternate approach for data abstractions based on explicit mappings from the domains of program variables to predefined abstract domains. The mappings are then "lifted" to abstract program operations to generate an abstract model. The abstract model so constructed is conservative with respect to the branching temporal logic fragment ACTL (the universal fragment of computational tree logic) [7]. Furthermore, the abstract model is exact for all properties expressible in CTL*, if the domain mappings are a congruence with respect to program operations [7].

In the next section a brief overview of model checking is given. Section 4 discusses the proposed approach for model checking of software. The use of weakest preconditions for performing control and data abstractions is described. The effectiveness of these techniques are illustrated using examples that arose in the analytic verification of a space craft controller that is being developed at NASA [16, 12]. Section 5 outlines the data abstraction technique based on explicit domain mappings.

A long-term spin-off of this technology is the development of light-weight, automated versions of these analytic V&V tools that continuously monitor a software system as it is executing. This technology has the potential to anticipate errors before they occur, thus enabling preventative measures to be invoked. For example, a model-checker could be synchronized to the execution of a software system, and explore in advance possible execution paths. Paths leading to safety violations, deadlocks, and other unanticipated concurrency bugs would be flagged and prevented (e.g., through run-time insertion of additional synchronization steps). To use these tools in a runtime monitoring mode requires that they be made extremely computationally efficient.

3 Analytic Verification Approaches: Model Checking

Model checking has proven very successful in debugging and verifying behavioral aspects of industrial-strength hardware applications and protocols [17, 8, 12]. This success can be largely attributed to the highly automated and algorithmic nature of model checking techniques that automate reasoning about error-prone concurrent applications. Unlike traditional, general-purpose verification approaches based on interactive theorem proving, model checking techniques focus on decidable fragments of logic over which automated reasoning is effective, without human guidance.

Typically, model checking is used to establish temporal properties of finite state transition systems. The finite state transition system is represented as a Kripke model and the properties are expressed in propositional temporal logic. Several variants of propositional temporal logic such as linear temporal logic, branching time temporal logic, and associated variants have been used to express temporal properties. Different model checking tools have been developed supporting these variants of temporal logic. These model checkers differ in the underlying representation of the finite state transition system that they employ and in the methods that they use to efficiently compute the reachable states of a transition system. An important problem that is pervasive in all model checking tools is the combinatorial explosion of the reachable state space even for moderately sized applications. This has been called the *state-explosion* problem. We discuss two broad approaches that have been previously pursued for dealing with the state-explosion problem and the associated model checking tools below.

Symbolic model checkers such as SMV [17] identify the state of the system with a predicate denoting the valuations of the variables of the system. The predicate encoding of a state can be naturally extended by expressing each transition as a relation between the current state variables and next state variables. The entire transition system can then be viewed as a conjunction of individual transitions. Symbolic model checkers like SMV use canonical representations to represent the formulae denoting a state, a set of states, and the transition relation. These specialized representations include binary decision diagrams (BDD), and binary moment diagrams (BMD) [2, 1]. A BDD can be thought of as first a binary decision tree for a propositional formula where each horizontal level is a boolean variable and the left and right branches represent the true and false valuations of a boolean variable. Each leaf of the tree is either true or false depending on whether the propositional formula is true or false under the valuations of the boolean variables repre sented by the path to that leaf. A binary decision diagram is a binary decision tree where common sub-trees have been merged, thus obtaining a directed graph. BDDs and BMDs often lead to a compact representation of the transition system by exploiting the sharing of structure in the sub-components of the system.

The reachable states of the system can be computed using a fixed point computation by either computing the *forward* images of a set of states at each iteration or by computing the *reverse* images of a set of states at each iteration. The convergence is guaranteed as the number of states is finite. Each temporal logic formula in a symbolic model checker is typically identified with the set of states in which it holds. The formula denoting the states satisfying a universal property (safety or invariant property) can be identified with the greatest fixed point of a monotonic functional denoting the set of the states satisfying the given property. The set of states satisfying an existential property (liveness or eventuality property) can be similarly characterized by the least fixed point of a monotonic functional. The computation of these fixed points can be iteratively done by starting with true (false) for the safety (eventuality) property [17].

Explicit state model checkers such as SPIN and Murphi, in contrast to symbolic model checkers, do not employ predicate encodings for representing the states of the transition system. The states are represented directly as a tuple of assignments to the variables of the system. In early explicit state model checkers the reachable states were precomputed by using the explicit representation of the transition system and the properties were then be checked over these reachable states. But such precomputation of the reachable states is not computationally tractable for even moderately sized applications due to large number of states. In order to circumvent this problem explicit state model checkers like SPIN and Murphi compute the reachable states *on-the-fly* along with each property. Several additional techniques are employed by these model checkers to reduce the number of explicit states that need to be enumerated. These techniques called *state-space-reduction* techniques are typically based on the notions of equivalences among states or paths. Murphi uses *symmetry reduction* techniques to avoid enumerating equivalent states and SPIN *uses partial-order reduction* techniques to avoid enumerating paths that lead to equivalent states.

4 Automatic Abstractions for Model Checking Software Systems

This section discusses the use of weakest preconditions in performing control and data abstractions. Automatic abstraction algorithms will generate an abstract model from a concrete program and requirements expressed as temporal properties to be verified. Semantic slicing based on weakest preconditions to perform control abstractions is described first. Subsequently, we discuss the use of weakest preconditions in performing data abstractions.

4.1 Automating Control Abstractions by Semantic Slicing

In many large scale software applications, often only certain portions of the application are relevant to a given property that is to be verified. The irrelevant portions of the program can be automatically discarded by using techniques similar to program slicing. This leads to a reduced program that is computationally tractable for analytic V&V. Instead of slicing the task programs with respect to variables, the programs are sliced with respect to state predicates appearing in the given temporal property. The reduced program that is generated from a given state predicate has the following guarantee: its error traces for the given property are in direct correspondence to the error traces of the original program.

Program slicing is a technique to extract a partial program that is equivalent to an original program over a subset of the program variables. The input to a traditional slicing algorithm is a program and a designated subset of variables, the output of the slicing algorithm is a partial program that for every program execution has identical values assigned to the designated subset of variables upon program termination. The key idea of slicing algorithms is to work backwards from the program end-point, keeping statements that have an effect on the designated variables and removing statements that have no effect. As the algorithm works backwards over the program statements, additional variables might be added to the designated set if they have occur in expressions which modify the designated set.

The concept of program slicing can be extended to abstract state-based programs for the purpose of model checking. However, instead of slicing with respect to variables, the programs are sliced with respect to state predicates, starting from a statement which contains the operation(s) which are required to only be executed in particular states. In other words, the programs are sliced with respect to an invariant. The partial program that is generated from such a property has the following guarantee: its error traces for the never property are in direct correspondence to the error traces of the original program. This is called WP-program slicing, as the algorithm is defined using Dijkstra's weakest-precondition calculus.

Two rules for this slicing calculus are described. **Slice** takes a code-seq terminated by a statement, and a state predicate, and returns a reduced code-seq. It determines where it can substitute the IDLE statement without affecting error traces by calculating when a statement " passes through" a state predicate. This occurs when the weakest precondition is the same as the state predicate. (The IDLE statements can later be removed if not needed for preserving the interleaving semantics. While slicing with respect to invariant properties, the statement can be discarded; whereas while slicing with respect to existential or liveness properties the IDLE statement is needed in order to preserve the possible interleavings.)

The slicing rule for straight-line sequences is:

```
Slice({code-seq; statement},state-predicate)
->
If WP(statement,state-predicate) =
   state-predicate
then
{Slice(code-seq,state-predicate);IDLE}
else
{Slice(code-seq,
       WP(state-predicate,statement));
 statement}
```

The rule states that IDLE can be substituted for a statement if the statement has no effect on the state predicate. Otherwise, the weakest precondition of the state-predicate is substituted for the state-predicate and slicing continues backwards. A more elaborate calculus would allow substituting a simpler statement that had the same weakest precondition.

The rule for conditionals requires the definition of a weakest predicate reduction. R is a *predicate reduction* of Q by P iff $R \vee P \Rightarrow Q$. R is a *weakest predicate reduction*(WPR) of Q by P iff for any S which is a predicate reduction of Q by P, $S \Rightarrow R$.

The rule for conditionals is defined on the following pattern:

```
Slice({code-seq; if P then then-statement
      else else-statement, Q})
Let then-WP = WP(then-statment, Q),
    else-WP = WP(else-statment, Q)
Let reduced-then-WP = WPR(P,then-WP)
    reduced-else-WP = WPR(not(P),else-WP)
```

Note that when the condition P implies then-WP, the IDLE statement can be substituted for then-statement. In other words, the then-statement is executed only in a context where it will not generate an error trace. Define reduced-then-statement as IDLE when P implies then-WP, otherwise it is the same as then-statement. Similarly, when not(P) implies else-WP, the IDLE statement can be substituted for else-statement. Define reduced-else-statement accordingly. The rule for slicing conditionals can now be defined:

```
Slice({code-seq; if P then then-statement
      else else-statement, Q} ->
{Slice(code-seq,(P and reduced-then-WP) or
                (not(P) and reduced-else-WP));
      (if P then reduced-then-statement
      else reduced-else-statement)}
```

Note that the weakest precondition for the conditional is composed of reduced weakest preconditions for the then- statement and the else-statement. A discussion of the application of these rules to task programs of a space controller executive are described in [16].

4.2 Overview of Automating Data Abstractions

Data abstractions play a crucial role in generating abstract models of software programs that are amenable for model checking. Typical software applications employ data types such as numbers, lists, trees, queues, stacks etc. that are often unbounded in advance. Simple translations of such applications leads to models with unbounded structures. Such models cannot therefore be analyzed by model checkers which operate on finite state models.

In the following two sections we describe two approaches for automatically generating an abstract model that is amenable for model checking starting from a concrete/program model. The first one is based on the generation of an abstract model based on weakest preconditions and is an extension of the approach described in [11]. The second approach is based on generating a set of mappings over the domains of the variables of the program and automatically lifting the program operations to abstract program operations. This approach extends research by Clarke et. al on abstractions in [7]. The theory developed previously by [7] dealt only with homomorphisms on the logical operators. Further, the domain mappings and the corresponding abstract logical operations are manually supplied that work.

The approach for performing data abstractions in [11] uses weakest preconditions to compute an abstract model from a program expressed in terms of a simple guarded command language. Even though it is conceivable that many software applications can be modeled or mapped in to such a simple language, often there is considerable overhead incurred in analyzing such a flattened representation of software applications. Often, high level control constructs employed in these applications such as loops and other concurrency primitives have to be rediscovered in order to effectively automate the generation of abstract models. Furthermore, the translation of applications from high level languages into such a formalism needs to carefully devised as naive translations may lead to state explosion. The main contribution of our work is first to extend the approach described in [11] to the richer control, data and concurrency constructs that occur in concurrent software applications. Second, we generate the abstract models compositionally in contrast to the approach in [11], where the application is always considered as a whole. This limits the ability of the approach in [11] to scale to large, complex applications. The extended approach is illustrated by generating an abstract model for the release property of the space controller resource manager verified in [12].

The basic assumptions underlying both these methods is the same. The notions of concrete and abstract program models used by these two approaches are formalized below.

Software/Concrete Model The semantic model representing the concrete program is a state transition system where each state is a valuation of a finite number of the program variables $V_1, \cdots, V_n$. over the types of the variable denoted by the corresponding domains $D_1, \cdots, D_n$. The state space of the program is bounded by $S = D_1 x \cdots x D_n$. The transition system T can be formally defined by the triple $\langle S, I, R\rangle$ where I, a subset of S, is the set of initial states and R, a subset of $S\ x\ S$, is the transition relation over states.

Abstract Model The abstract model of T, T_{abs}, is a transition system denoted by the triple $\langle S_{abs}, I_{abs}, R_{abs}\rangle$. T_{abs} is defined in terms of an abstraction function $F_{abs} : S \rightarrow S_{abs}$. The abstraction function F_{abs} is naturally extended to a set of states S', denoted by $F_{abs}(S')$ where $F_{abs}(S') = \{F_{abs}(s') \mid s' \in S'\}$. Similarly, the abstraction function F_{abs} is naturally extended to the relation R, denoted by $F_{abs}(R)$, where $F_{abs}(R) = \{F_{abs}(s_1, s_2) \mid (s_1, s_2) \in R\}$. The following conditions must hold for the abstraction function F_{abs}.

1. $F_{abs}(I) \subseteq I_{abs}$.
2. $F_{abs}(R) \subseteq R_{abs}$.

If conditions 1 and 2 are equivalences, then the abstract model is *exact*. Any invariant property that is true of the abstract model is true of the program, but not vice-versa. The converse also holds when the abstraction is exact.

The abstract model obtained using the first approach has the guarantee that whenever an invariant property is true in the abstract model then it is true in the concrete model, but not vice-versa. The second approach is more general and preserves truth values of the abstract model for more expressive class of properties including CTL and universally quantified fragment of CTL* [7].

4.3 Generating Data Abstract Models Using Weakest Pre-Conditions

The key idea in this approach is to identify a set of control predicates. The abstract model can then be regarded as recording the control predicate values as concrete transitions are executed. The resulting abstract state machine is defined in terms of the valuations of these predicates and hence is finite state and can therefore be analyzed by a model checker.

Given a set of control predicates $P = \{p_1, \cdots, p_n\}$, the state space of the abstract model is a lattice induced by predicates in P. The elements of the lattice are conjunctions of atomic literals over P with the additional element false. The lattice is complete with the implication ordering relation with the join and meet being given by conjunction and disjunction respectively. The state space is bounded with at most $3^(n+1)$ elements.

The concrete transition relation R is viewed as a finite set of labeled transitions corresponding to the program statements. Recording the values of control predicates is done by computing the successors for each abstract state with respect to all of the concrete transitions whose labels correspond to the program control being at that point and which are enabled. The absence of successors for an abstract state s_1 with respect to a transition with label t_1, is denoted by setting the abstract successor state to false. The rule for computing the abstract successor state for a concrete assignment statement is given below.

```
succ(s1, x := e)[i]  =
{true,  if s1 => WP(x := e, pi
 false, if s1 => WP(x := e, not(pi))
 True   Otherwise}
```

The i^{th} component of the abstract successor state in the above rule is computed based on the i^{th} control predicate. The component is true if the weakest precondition of the assignment statement with respect to the corresponding control predicate follows from the current abstract state. It is false if the weakest precondition with respect to the negation of the predicate is implied by the current abstract state. It evaluates to true otherwise.

A Simple Example of Abstract Model Generation Consider the following simple program with two processes P1 and P2 with the global shared variable x. L1 and L2 are the labels associated with the two assignment statements in the processes. The initial value of the variable x is set to 1. This is denoted by the assignment in the initially block. The two run statements result in the simultaneous spawning of the processes P1 and P2.

```
Concrete Model/Program

Integer x;
Process P1();
   L1: x := x + 1;
end;
Process P2();
   L2: x := x + 2;
end;
Main Program
    Initially
      x := 1;
    end;

    begin
    run P1();
    run P2();
    end;
end.
```

An invariant property that can be established from this program is `x > 0`. The abstract model can be generated from the above concrete program by the following sequence of steps. WP below denotes the weakest pre-condition.

1. Let the control predicate `P = x > 0`.
2. The initial abstract state `S0 = x > 0` since the initial value of `x = 1` and `(x = 1) => (x > 0)`.
3. Abstract Transition Relation:
   (a) `(S0, L1) = S1: x > 0 since WP(l1, x > 0) = x > - 1 and (x > 0) => (x > -1).`
   (b) `(S0, L2) = S2: x > 0 since WP(l2, x > 0) = (x > - 2) and (x > 0) => (x > -2).`
   (c) `(S1, L2) = S3: x > 0 since WP(l2, x > 0) = x > -2 and (x > 0) => (x > -2).`
   (d) `(S2, L1) = S4: x > 0 since WP(l2, x > 0) = x > - 1 and (x > 0) => (x > - 2).`
   In the above generation of state machine, for illustrative purposes, no form of state equivalence has been assumed. Steps c) and d) can be eliminated by identifying the abstract states `S2` and `S1` with the absstract state`S0`.
4. The above state machine can be used by a model checker to establish the invariant `x > 0`. In this case, the reachable states is `S0` with `x > 0`.

A non-trivial example : Remote Agent Executive of a Spacecraft Controller In this section we illustrate how the same technique can be applied to a nontrivial example of a resource manager of a space craft controller. The resource manager was verified using the SPIN model checker in [12] and several bugs were found. An invariant property that was found to be incorrect there concerned the release of locks by the task processes whose resource usage is monitored by the resource manager. We outline how the abstract model exhibiting an error trace for this property can be automatically generated using the invariant property. A very brief description of the Remote agent executive, the invariant property and the relevant aspects of the concrete model are first described. Subsequently, we illustrate how the proposed approach can be used to automatically generate an abstract model which can be analyzed by a model checker to generate a counterexample to the invariant property. For a complete description of the Remote agent executive the reader may refer to [12].

The remote agent executive monitors a set of onboard flight tasks that operate on a common set of shared resources. Each task needs specific resources and also requires certain properties of the resources to hold while it is executing. The current set of properties of a resource are stored in a global database Db. In order to execute, each task first locks (*snarfs*) the required resources, then the task tries to achieve the properties of the resources that must be maintained while it is executing. If both these are successful the task may go ahead and performs its computation, releases the acquired resources and successfully terminates. Mutual exclusion among shared resources is achieved by locking and the resource locks that are held by a task process are maintained in a global lock table, *Locks*.

In addition to the task processes two predefined processes that execute in the remote agent executive are the environment *Env*, and a daemon process, *Daemon*. The daemon process is periodically woken up in response to certain events signaled by the task and the environment processes and checks whether the properties associated with a resource in the database and lock table are consistent. If this is not the case, then all the task processes that are in violation are aborted by the daemon. The environment process in response to external stimulus (not modeled here) may change the properties associated with a resource in the global database. On a change, the process signals the daemon to perform consistency check of the properties. An important invariant that needs to be established of this model is that any task once it successfully terminates releases all its resources. This can be modeled in terms of a predicate called issubs(Taskid, resource, property, Locks), which checks whether a task with Taskid is holding the resource. The predicate should evaluate to false once the task successfully terminates.

This is modeled as an in-line assertion in the concrete model described below.

```
Concrete Model/Program

List Db of (Resource, Property);
List Locks of (Resource, Property, Subscribers);
List activetasks of Taskid;

Process task(Taskid, Resource, Property);
begin
 {atomic
  {snarf_property (Taskid, Resource, Property);
   achieve_property(Taskid, Resource, Property)}
   perform_computation();
 } unless {active_tasks[Taskid].status = abort};
 active_tasks[Taskid].status :=  Terminate;
 {Release_locks(Taskid, Resource, Property)}
   unless {active_tasks[Taskid].status = abort};
 assert {
 (not(issub(Taskid,Resource,Property,Locks));}
end;

Process Daemon();
List Taskids of Taskid;
begin
 If(Daemon.status = Wakeup) then
   Taskids = Db_violated();
   While(Taskids <<> nil) do
    active_tasks[head(Taskids)].status := aborted;
    Taskids = tail(Taskids);
   end;
end;

Process Env(Resource, NewProperty);
begin
 Db[Resource] := NewProperty;
 Daemon.status := Wakeup;
end;
Main Program
begin
 activetasks = Db = Locks = nil;
 run Daemon();
 run task(Taskid, resource, property);
 run Env (resource, NewProperty);
end.
```

In the concrete model above, " atomic" indicates an indivisible sequence of statements whose execution is not be interleaved with any other statements in other processes. Exceptions are handled by the special construct "unless". $\{S_1; \cdots, S_n\}$ unless (cond) has the semantics that statements are $S_1, \cdots, S_n$ are executed sequentially as long as cond evaluates to false. Otherwise, the program control jumps to the statement following the unless statement. Therefore, in the above example, a task locks resources, achieves the desired properties of the resources and performs its computation as long as the it is not aborted by the daemon due to an inconsistency between the database and the lock table. Such an inconsistency could be triggered by the environment changing the database. In order to automatically compute an abstract model corresponding to the above given concrete model we first describe rules for computing the abstract successors for additional concrete model constructs such as conditionals, repetition and unless statements below. The partial abstract model generated for each of the processes based on these rules is given next.

– *Conditional Transition*: There are two cases to be considered here. First, when the statement is enabled i.e. the condition *cond* is implied by the current abstract state. In this case the componentwise computation of the abstract successor state is given below.

```
succ(s1, if(cond) x := e)[i] =
{true, if s1 => WP(if(cond)x := e,pi) &
          s1 => cond,
false, if s1 => not(WP(if(cond)x := e,pi)) &
          s1 => cond,
True    Otherwise if s1 => cond}
```

If the statement is not enabled then there is no abstract successor and this denoted by the abstract successor state being assigned the value false.

```
succ(s1, if (cond)x := e) =
{false if  s1 => not(cond)}.
```

– *Repetition Transition*: An abstract successor for a repetition statement of the form: while b do S can be computed in terms of the loop invariant I. Given invariant I, the repetition statement is split into two actions with the control points b and I, not(b) and I corresponding to the control being in the loop or going out of the loop respectively. The abstract successors for each of these cases is given below.

```
succ(s1, b and I S)[i] =
{true  if s1 => (b and I => pi)
 false if s1 => not((b and I => pi))
 True Otherwise}.

succ(s1, not(b) and I S)[i] =
{true if s1 => (not(b) & I => pi)
 false if s1 => not((not(b) & I => pi))
 True Otherwise}
```

– *Assertion Transition:* The underlying semantics of an assertion statement is that whenever the program control reaches such an expression it can progress to the subsequent statement only when the expression evaluates to true. Otherwise, the program control is said to be blocked. For the non-blocking case the computation of the abstract successor state is given below.

```
succ(s1, exp1 = exp2)[i] =
{true, if (s1 => (exp1 = exp2)) &
           s1 => pi
false, if (s1  => (exp1 = exp2)) &
           s1 => not(pi)
True   Otherwise}.
```

The blocking case leads to no abstract successors as specified below.

```
succ(s1, exp1 = exp2) =
{false  if (s1 => not(exp1 = exp2))}
```

– *Unless Transition*: Unless is a special construct for handling exceptions. The typical usage is of the form : $(S_1;\ \cdots;\ S_n)$ unless (cond). Before the execution of each statement S_i in the sequence, the condition cond is checked and the next statement is executed only if cond evaluates to a false. Otherwise, the rest of the statements are ignored and the program control jumps to the statement following the unless construct. For computing abstract successors, the unless construct can be transformed into a series of conditional statements of the form: $(if(not(cond)\ S_1);\ \cdots;\ \ if(not(cond)\ S_n)$. These statements can be handled as conditionals.

The abstract model based on these rules can be computed by the following sequence of steps. In addition to the invariant two additional control predicates isconsis and isnotabrt are used. These denote the consistency of the database and the lock table and the status of a task not being aborted respectively.

1. The control predicates are:

```
P1:not(issub(Taskid,resource,property,Locks)),
P2:isconsis(Db, resource,property, Locks),
P3:isnotabrt(this,activetasks)
```

2. The task process abstract machine:
   (a) `S0 = P1, P2, P3`
   (b) `S1a = S0, action incompatibility check in snarf property = S1 = (P1, P2, P3)`
   (c) `S2a= S2a, append action in snarf property = (not(P1), P2, P3)`
   (d) `S3a= S2a, achieving property = S2a`
   (e) `S4 = S2a, terminate action = (not(P1), P2, P3)`
   (f) `S5 = S2a, action release lock procedure = (P1, P2, P3)`

   The environment process abstract machine:
   (a) `E0 = (True, True, True)`
   (b) `E1 = E0, Changing the Db = (True, not(P2), True)`

   The daemon process abstract machine:
   (a) `D0 = (True, True, True)`
   (b) `D1 = (D0, check-locks) = (not(P1), True, True)`
   (c) `D2 = (D1, lockviolation) = (not(P1), not(P2), True)`
   (d) `D3 = (D2, abort task) = (not(P1), not(P2), not(P3))`

Composing these three machines gives the entire abstract machine. Composition needs to be carefully done since the actions suffixed by "a" denote atomic actions. The composed abstract state machine contains the following sequence of transitions which can be analyzed by a model checker to give the following counter-example:

```
S5 x E0 = (not(P1), True, P3) ->
E1 x D0 = (not(P1), not(P2), P3)  ->
D2  -> D3 = (not(P1), not(P2), not(P3))
```

which is the violating state. The above counterexample corresponds to the subtle interleaving that arises due to the environment changing the database and the daemon aborting the task after the task has finished its computation but before it has released its locks.

4.4 Generating Abstractions Using Explicit Homomorphisms

In the approach based on the generation of abstract models using explicit homomorphisms the abstract transition system T_{abs} over the program variables $V_1, \cdots, V_n$ over the abstract domains $D'_1, \cdots, D'_n$ is obtained by using a family of onto mappings $h_i : D_i \rightarrow D'_i$ where D_i denotes the concrete domain. Note that h_i is an equivalence relation over D_i where $d_i =_{abs} d_j iff h_i(d_i) = h_i(d_j)$ for any two elements d_i and d_j of D_i. This is naturally extended to a mapping h over the states by pointwise applications of h_i. The characterization of an exact abstract model can be given in terms of the family of mappings h and the primitive operations P. An abstract model is exact iff h is a congruence with respect to each of the primitive operations in the program. If $(h(d_i) = h(e_i))$ then $P(d_1, \cdots, d_n) = P(e_1, \cdots, e_n)$ for $1 \leq i \leq n$.

The naive way of generating the abstract model from a given program model T and a set of onto mappings is infeasible for software since it requires an explicit representation of the program model T and often the program model T is either too large or is unbounded. A typical way around is to generate an approximation of the abstract model from the text of the program itself by doing a form of abstract compilation. This is usually done by identifying the set of "primitive operations" in the program and symbolically simulating the program starting from the initial state, and executing the program. The transition system T can be obtained in this way as well as T_{abs}. The only difference is in the interpretations associated with the primitive operations. If we interpret the primitive operation more abstractly then we get T_{abs}.

So, for automatic generation of the abstract model T_{abs} there are three issues that need to be addressed:

1. Generating h automatically.
2. Showing that h is a congruence with respect to primitive operations P.
3. Deriving abstract versions of the primitive operations from P and h ?

The proposed approach is to identify some useful classes of h and choose dynamically from among these based on the property that is being attempted and other user supplied predicates. This is done in terms of the following steps.

1. For a given set of predicates and identify the domain mappings using a predefined classification hierarchy.
2. Check whether this mapping is a congruence with respect to the primitive program operations.
3. If so, "lift" all the program operations to corresponding operations over the new domain. Otherwise, the abstraction fails.
4. Perform abstract compilation with respect to the new predicates and the domain.

5 Related Work and Conclusions

The main challenge in using automated V&V tools such as model checkers is in coming up with a tractable finite state model i.e., dealing with the state explosion problem. A lot of work has been done in combating the state explosion problem for hardware applications [4, 5, 6]. Several techniques such as symmetry reduction [18], partial-order red uction [9, 14] and symbolic encodings of states using representations such as *BDDs, BMDs* [17, 1, 2] have been considered. However, these techniques only work on finite models. They can not be applied in general to software systems, which typically use data types operating on unbounded domains and hence have models which are not bounded in advance.

The use of data abstraction in model checking to handle some limited types of infinite state systems are discussed in [19, 7, 10, 15]. In [19], Wolper uses the notion of *data independence* to transform certain temporal properties specified over infinite data values to equivalent properties specified over a finite domain. In [7], Clarke et. al, discuss how user defined abstractions can be used for model checking. However, no attempt is made there to automate the generation of abstractions. Two approaches that address aspects of automatic generation of abstractions are [15, 11]. Jackson in [15] discusses how certain properties involving unbounded sets and predicates over such sets can be reduced to equivalent properties over bounded sets. In [11], the interactive theorem prover PVS is used to generate an abstract model. However this approach is limited in terms of the data, control, and concurrency abstractions and in the size of the abstract model to be used for software applications. We have shown how this approach can be extended to handle typical data concurrency and control abstractions that frequently occur in software applications. The use of weakest preconditions for program slicing is also discussed in [3]. However, the approach described there is applicable only to sequential programs.

References

1. R. E. Bryant, and Y.-A. Chen, "Verification of Arithmetic Circuits with Binary Moment Diagrams", *In 32nd ACM/IEEE Design Automation Conference (DAC)* 1995.
2. R. E. Bryant, "Graph-based algorithms for boolean function manipulation", *In IEEE Trans. on Computers, Vol. C-35, No. 8.* 1986.
3. J. J. Comuzzi and J. M. Hart, "Program Slicing using Weakest Preconditions", *in Proc. FME: Industrial Benefit and Advances in Formal Methods, 3rd Intl. Symposium* , LNCS 1051, 1996.
4. *5th International Conference on Computer Aided Verification*, CAV-93, Springer Verlag LNCS 697.
5. *6th International Conference on Computer Aided Verification*, CAV-94, Springer Verlag LNCS 818.
6. *8th International Conference on Computer Aided Verification*, CAV-96, Springer Verlag LNCS 1102.
7. E.M.Clarke, O. Grumberg, D.E. Long , " Model Checking and Abstraction", *In Proc. of 19th ACM Symp. POPL* 1992.
8. A. Th. Eiriksson and K.L. Mcmillan, "Using formal verification/analyses methods on the critical path in system design : A case study," *In Proc. of Computer Aided Verification*, LNCS 939, Springer Verlag 1995.
9. P. Godfroid, " Using Partial Orders to improve automatic verification methods," *In Proc. of 2nd Workshop on Computer Aided Verification* 1990.
10. S. Graf, "A tool for symbolic verification and abstraction," in CAV93.
11. S. Graf, H. Saidi, " Constructing Abstract Graphs Using PVS," in CAV96.
12. K. Havelund, M. Lowry, and J. Penix, " Formal analysis of a space craft controller using Spin", *NASA Ames Research Center Technical Report*, 1997.
13. K. Havelund, T. Pressburger, " Translating Java to Spin," Manuscript Under Preparation, NASA Ames Research Center.
14. G. Holzmann, D. Peled, " The State of SPIN," in CAV-96.
15. Jackson, D.E., " Abstract Model checking of infinite specifications," *In Proc. of Formal Methods in Europe* 1994.
16. M. Lowry, K. Havelund, and J. Penix, " Verification and validation of AI systems that control deep-space spacecraft.", *In Proc. of ISMIS* 1997..
17. K. McMillan, " Symbolic Model Checking", *Kluwer Academic Publishers*, 1993.
18. C. Norris Ip, " State Reduction Methods for Automatic Formal Verification," Ph.D. Thesis, Department of Computer Science, Stanford University, December 1996.
19. Wolper, P., " Expressing Interesting Properties of Programs in propositional temporal logic" *In Proc. 13th ACM POPL* 1986.

This article was processed using the LaTeX macro package with LLNCS style