Model Checking Software via Abstraction of Loop Transitions

Natasha Sharygina[1] and James C. Browne[2]

[1] Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA, USA 15213
nys@sei.cmu.edu
[2] The University of Texas, Austin, TX, USA 78712
browne@cs.utexas.edu

Abstract. This paper reports a data abstraction algorithm that is targeted to minimize the contribution of the loop executions to the program state space. The loop abstraction is defined as the syntactic program transformation that results in the *sound* representation of the concrete program. The abstraction algorithm is defined and implemented in the context of the integrated software design, testing and model checking. The loop abstraction technique was applied to verification of NASA robot control software. The abstraction enabled model checking for realistic robot configurations where all other state space reduction approaches, including BDD-based verification, predicate abstraction and partial order reduction, failed.

1 Introduction

Formal verification by model checking has the potential to produce a major enhancement in software reliability and robustness for software systems. The applicability of model-checking to software systems is severely constrained by "state space explosion". Data abstraction is a principal method for state space reduction [1, 4, 6, 13, 14, 16]. Predicate abstraction [7] is one of the most popular and widely applied methods for systematic abstraction of programs. Predicate abstraction is based upon abstract interpretation [5]. It maps concrete data types to abstract data types through predicates over the concrete data. However, complete predicate abstraction may be intractable due to its computational cost. Generation of a full set of predicates is typically infeasible for large programs.

All forms of abstraction may introduce unrealistic behaviors (behaviors not found in the concrete program) into the abstract program. Error traces from model checking of the abstract program are often used to rule out unrealistic behaviors. Excessive abstraction may introduce additional behaviors which result in state space explosion when attempting model checking for the abstract program. These drawbacks for general abstraction methods coupled with the potential effectiveness of abstraction, motivate research into targeted abstractions which can be applied selectively.

This paper formulates and evaluates an abstraction algorithm that minimizes the contribution of the loop executions to program state space. The loop abstraction generates an abstract program with the same static task graph as the concrete program from which it is derived but which specifies a minimum (or nearly minimum) number of traversals of the loops of the static task graph. These abstract programs have orders of magnitude smaller state spaces that the concrete programs from which they are derived. We demonstrate that the algorithm is *correct* in that the "abstract" program is a conservative approximation of the "concrete" program with respect to the control specifications of

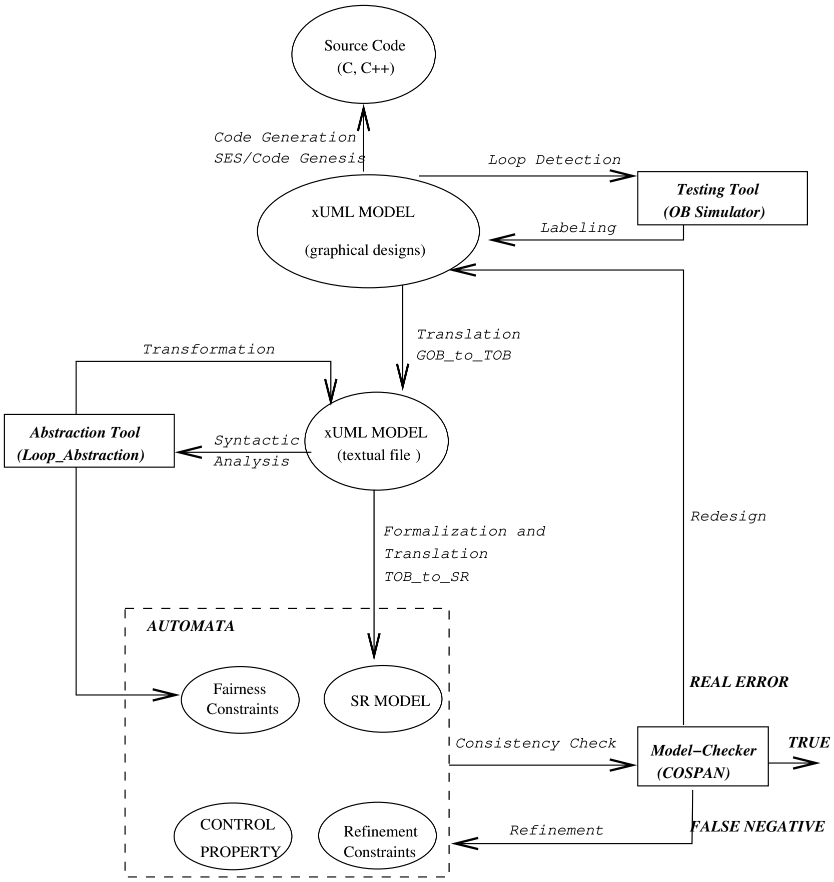

Fig. 1. The Integrated Design and Model-Checking Software Development Environment

the program. The correctness result implies that a control specification holds for the original program if it holds for the abstract program. Some loss of precision of data computations introduced by the abstraction is traded for the ability to conduct *practical* verification of behavioral specifications of control algorithms. The potential usefulness of loop abstraction is enhanced by the facts that control software are the obvious candidate systems for model checking to improve reliability and that almost all control systems implement feedback loops.

The properties of the loop abstraction algorithm are:

- It is computationally simple and requires storage linear in the size of the program since it is a source to source transformation based on static analysis of the program.

- It is based on syntactic manipulation of expressions, and produces a reduced program and therefore, it can be applied without change to the verification tool or the verification algorithm.

- It produces a syntactic representation of the abstract program and thus other model-checking state space reduction techniques, such as symbolic model-checking and partial order reduction, can be applied to the abstract program.

The loop abstraction algorithm has been implemented in the integrated high-level design (xUML) and automata-based model checking software development framework (**Figure 1**) and has been evaluated during verification of a NASA robot controller. It has been found to give order of magnitude reduction in the complexity and computational resource requirements for model-checking of control properties of a robot control system. Most importantly, the loop abstraction enabled completion of model checking for realistic robot configurations where all other approaches, including predicate abstraction [1, 15], failed.

Contents of Paper. Sections 2 defines a framework for the project. Section 3 defines the program syntax and semantics. Section 4 presents the loop abstraction algorithm. The effectiveness of loop abstraction is demonstrated in Section 5 that shows the verification results of the NASA robot controller system. Section 6 summarizes the paper and gives an overview of related work.

2 Integrated Design and Verification Framework

The loop abstraction algorithm has been defined in the context of the software development framework that integrates xUML modeling[3], testing and automata-based model-checking as shown in **Figure 1**. We refer the reader to [19, 22] for a detailed description of the integrated environment.

The framework consists of the following components:

1. *The xUML graphical specification and validation environment as it is implemented in the commercial tool,* SES/ObjectBench (OB) [17].

An xUML program is a set of interacting objects. The behavior of each object is implemented as a *Moore state machine* with a bounded FIFO input queue for events. The objects interact by sending and receiving events. Each state of the state machine which can receive an event is given a unique label. A sequential action is associated with each labeled state. Each action assigns values to state

[3] xUML is a dialect of UML with executable semantics. Programs written in xUML are *design level* representations which can be executed directly through discrete event simulation or interpretation and/or compiled to procedural source code. xUML is fairly widely used for development of control systems [11, 17]. A complete specification of the xUML notation can be found in [20, 21].

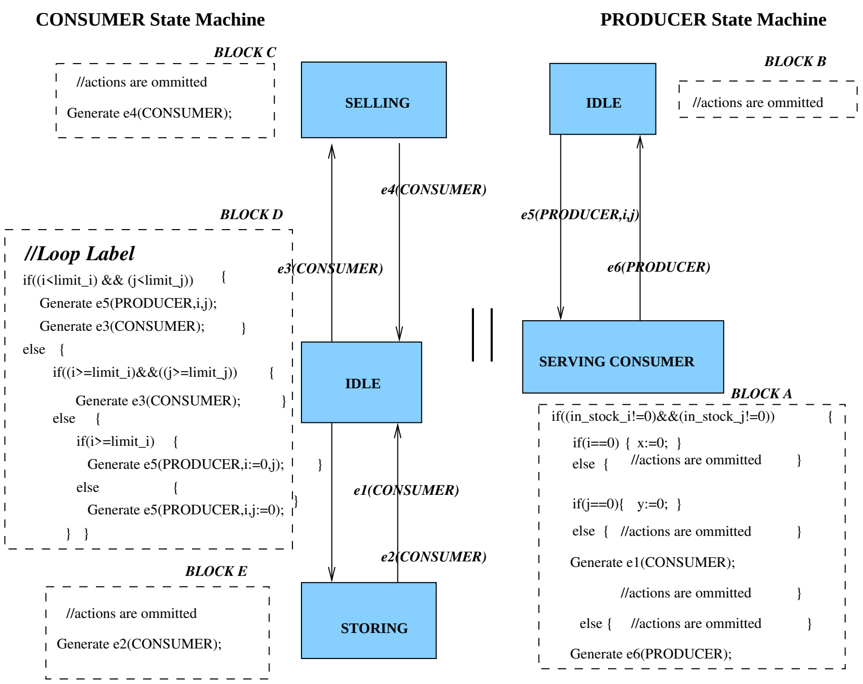

Fig. 2. The *Consumer-Producer* xUML Program

variables and generates events to be posted to its own input queue or the input queues of other state machines. The actions execute in run to completion mode. The action language for the implementation of xUML is a C-based language extended by the event generation and state machines manipulation commands. The execution model for an xUML system is asynchronous interleaved execution of the action language programs associated with the labeled states of the state machines. An example of the xUML system is given using a *Consumer-Producer* xUML program (**Figure 2**). The sample program is modeled by xUML state machines, representing behavioral specifications of the *Consumer* and *Producer* xUML objects.

Each state machine is represented as a collection of *blocks* that are activated by events. For example, *Block D* of the *Consumer* state machine represents a block of actions that can be activated by an input event *e4* or *e2* and labeled by the update of a variable *status := IDLE* (the label variables update statements are implicitly implemented by the xUML graphical development environment.) The activation of the block is followed by the execution of local statements for variables update and generation of output events *Generate e3(CONSUMER), Generate e5(PRODUCER,i,j), Generate e5(PRODUCER,i:=0,j)* and *Generate e5 (PRODUCER,i,j:=0)*. Note, the distinction between fields of the event *e5*: different data is passed by the event depending on the satisfaction of the specified conditions. For example, if at some point during the program execution a variable i is larger or equal to some predefined value, *limit_i*, than the event *e5* will pass a zero value to the *Producer* process, using the first supplemental data field of the event command.

2. *The* LOOP_ABSTRACTION *program.* This component of the integrated environment is the subject of the paper. The LOOP_ABSTRACTION program implements the loop abstraction algorithm which is outlined in Section 4. The *loop_abstraction* program takes as an input results of the program behavioral analysis conducted using the discrete event simulator. The event simulator is a part of the xUML specification and validation environment. During the simulation the program is executed by traversing possible *event sequences* which can arise from the execution of interacting xUML state machines. The set of actions that are repeatedly initiated by some event are manually annotated with a *Loop Label* in the xUML specification environment. An example of the annotation is shown in **Figure 2**, *block D*.

The *loop_abstraction* algorithm results in the syntactic program transformation of the original program that maps all traversals of a loop in the program control flow defined by different values of the program variables to traversals with values of *path_selection* variables (see Section 4.2) which abstract values uniquely specify each distinct event sequence within the loop.

3. *The automata-based model-checking tool,* COSPAN[4]. Consistency check is performed over the abstract SR model (SR is the input language of COSPAN) automatically derived from the abstract xUML program with respect to the specified control property, the set of fairness constraints and the approximation restrictions[5]. The following features provided by COSPAN are used to support the loop abstraction procedure:

- the *assume/guarantee mechanism* of COSPAN is used to add fairness constraints and refinement assumptions to the model-checking process.

- the *localization reduction* algorithm, automatically invoked by COSPAN during model-checking, is used to eliminate from consideration variables that do not effect the verification property.

3 Background

3.1 Program Syntax

Control software systems are often constructed as *compositions of sequential programs* which interact through sending of events, $S = p_1 \parallel ... \parallel p_n$. Each program, p = *(X,E,I,B)* is defined as a set of variables, X, a set of events, E, an initial

[4] A detailed description of COSPAN and its features can be found in [8, 13].

[5] The abstraction procedure uses a translator [22] that automatically transforms the xUML programs from the Graphical OB representation into SR, an input language of the model-checker, COSPAN. Specifically, the LOOP_ABSTRACTION program is applied to the intermediate representation of the translation result, the textual representation of the xUML programs.

condition, I, and a set of *basic blocks* (defined next), B, that contain commands that modify the program variables, and send and receive events. For example, an *assignment* command, $x := any\{\ exp_1,\ldots,exp_n\ \}$ is a *non-deterministic* assignment, after which a program variable, x, will contain the value of one of the expressions $exp_1,\ldots,\ exp_n$; *'Generate e(ID,exp)'* is a *communication* command that sends an event, e with some data, *exp*, to the destination program identified by its name, *ID*[6].

Definition 1 [Basic Block] (cf. [9]): *A* basic block *is a sequence of statements for which execution can be initiated only through the statement at the head of the block and which, once initiated, executes to completion.* Execution of a basic block is initiated by arrival of an event.

Events are distributed via FIFO queues, one queue for each sequential program. The execution model for a *sequential program* is: a) An event arrives in the input queue of a sequential program and some basic block of the program is enabled for execution in "run to completion" mode. b) The enabled basic block is executed. c) Execution of a basic block may result in events being sent to the program containing the executing basic block or to other programs. d) At the end of the execution of a basic block the program halts and awaits arrival of its next event.

Definition 2 [Output of a Basic Block]: *The* output of a basic block *is an event or sequence of events.* The output from an instance of the execution of a basic block is determined by the control structure within the block. Each instance of the execution of a basic block is a traversal of the tree determined by the control structure. The control statements which generate the tree will be referred to as the *block output guard.* The outputs of a basic block are determined by the leaves which are reached in the execution of the block. Thus, each branch of the block output guard controls *one* output of a block.

Figure 3 illustrates the concept of the basic block. The control flow graph

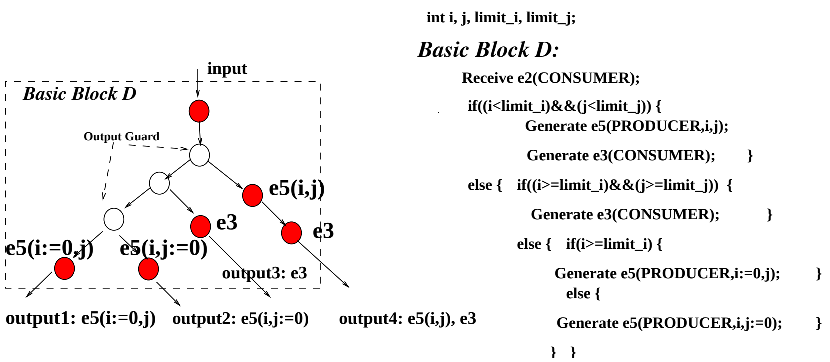

Fig. 3. Demonstration of a Basic Block Concept

[6] For the complete list of the commands see [17]

(at the command level) illustrates the control flow paths that determine the outputs of the basic block.

The execution model for the system is *asynchronous interleaved* execution of the basic blocks of the sequential programs. a) One program from among those which are enabled for execution (those programs with events in their input queues) is non-deterministically selected for execution. b) The basic block in the selected program which consumes the event at the head of the event queue is executed and step *a* is repeated.

Definition 3 [Basic Block Control Flow Graph]: *The nodes of* the basic block control flow graph of the system are *basic blocks* of the composing sequential programs. *The arcs* of the basic block control flow graph of the system connect basic blocks which are the sources and targets for *events.* Therefore a control flow graph can also be specified as generation and consumption of a sequence of events.

The control flow properties of the system behavior can be stated in terms of control at the basic block level by referring to events that initiate execution of basic blocks.

This works exploits the *atomicity* of the program executions to identify loops of the program execution (loops across basic blocks).

Definition 4 [Loop]: *A* loop *in a basic block control flow graph of a system is defined by a repeated execution of a path which begins with the generation of a unique event by a basic block and ends at that same basic block (loop basic block,* B^{loop}*).*

Each loop is guarded by a set of the basic block output guards of the loop basic block, B^{loop}, and their dependence set, B^{depend}. The dependence set is the set of basic blocks which output guards operate on variables that are dependent on the output guard variables of the loop basic blocks. Let us call the variables of the basic block output guards that define the loop, *loop control variables.*

3.2 Program Semantics

The syntax of the program defined above can be given an execution semantics as an asynchronous transition system (ATS) [10] composed of finite state machine interacting through finite, non-blocking FIFO queues.

Definition 5 [Event Queue]:(cf. [10]) *An* event queue, Q_i = *(V,N,E,L)* is defined by the *the queue vocabulary*, V, by the *size of the queue*, N, by the *vector of events stored in the queue*, E, and the *content of the stored events*, L, defined as a finite set of the values. The values are expressions on the system variables, or constants. For a set of queues, $\mathcal{Q}$, the queues vocabularies are disjoint.

Definition 6 [Finite State Machine]:(cf. [10]) *A* state machine, *M, is defined as a tuple,* $M = (X,S,s_0,I,O,\mathcal{Q},T)$, where

- *X is the finite set of variables*;
- *S is the finite set of possible binding of values to X*;
- s_0 *is* an element of S, *the initial state*;
- *I is the set of input events*;
- *O is the set of output events*;

- $\mathcal{Q}$ *is a set of event queues*;
- T *is the transition relation specifying the allowed transitions among S.*

Definition 7 [Trace of a State Machine]: *An infinite sequence of states* $tr = s_0 s_1 ... s_n$*,* is a trace of FSM *if (1)* s_0 *is an initial state and (2) for all* $0 \leq i < n$*, the state* s_{i+1} *is a successor of* s_i*.*

Definition 8 [Asynchronous Transition System (ATS)]:(cf. [10]) *An* ATS *is a composition of finite state machines which interact by sending and receiving events.* The global state space is the product of the local state spaces of the composed state machines, the system event queue is the union of the sets of the queues of the separate machines, and the global transition relation is the union of the local transition relations.

Definition 9 [Trace of an ATS]: *The trace of an ATS is an interleaving of states from the traces of the state machines which compose the system.* The ATS may be constrained by **fairness constraints** that determines which traces of the model are confronted with the specification during model checking. If a fairness constraint is defined as set of states, then a *fair trace* must contain an element of each fairness constraint infinitely often.

Definition 10 [Refinement]: Let A and C be two ATS instances. Let L(A) and L(C) be the language of all traces from execution of A and C.

If $X^C \subseteq X^A$, and L(C) $\subseteq$ L(A) then C *weakly refines* A, C $\leq$ A.

Definition 11 [Control Refinement]: Let us define an operator R which projects from L(C) and L(A) all states which do not receive events. Call R.L(C) and R.L(A) *control traces* of an ATS.

If $X^C \subseteq X^A$ and R.L(C) $\subseteq$ R.L(A) then C *weakly refines control* of A.

Control refinement is defined to show behavioral correspondence between the control (event) sequences of the abstract and concrete programs. The program actions are grouped into basic blocks (as defined earlier) and execute in run to completion mode. Therefore R.L(C) and R.L(A) correspond to the basic block control flow graphs of systems C and A and L(C) and L(A) correspond to the traces of systems C and A.

Definition 12 [Control Property]: *A* control property *is a linear temporal logic specification defined over states that input events.* For example, in Fig.2 an event *e1(CONSUMER)* accepted by the *CONSUMER* program defines a control state of the program ATS.

4 Loop Abstraction

The execution behaviors of control software systems are typically dominated by cycles implementing feedback loops. The structure of the control flow graph is usually determined by a small set of variables (control flow variables). The paths in the control flow graph of a program with loops are usually determined by conditional statements (guards) which depend on a subset of the control flow variables (loop control variables). Model checking of such systems generates a traversal of the loops in the control flow graph for each possible value of each loop variable. Each traversal of the loop with different values of the loop control variables is distinct in the state graph of the program. Additionally each traversal of a loop will typically involve many variables ("don't care" variables) which do not participate in determination of the paths through the control flow graph. But each execution of a loop with different values for the "don't care" variables is also distinct in the state graph generated by the model checker.

Control flow properties (such as guarantee that the program components will never execute an unsafe sequence of control actions) are dependent only on the control flow graph of the system and are independent of the number of traversals of the loops of the control flow graph. Therefore the control properties of the concrete program can be model checked by model checking of an abstract program with the same control flow graph.

The abstraction presented in this paper generates an abstract program with the same static task graph as the concrete program from which it is derived but which specifies a minimum (or nearly minimum) number of traversals of the loops of the static task graph. The values of the "don't care" variables can also be freed in the abstract program. These abstract programs typically have orders of magnitude smaller state spaces than the concrete programs from which they are derived.

4.1 Sketch of the Loop Abstraction Algorithm

The algorithm iteratively analyzes and transforms basic blocks that define control flow within a loop[7]. The algorithm computes a number of outputs for each block that is executed in the control loop and uses the computed data to abstract the events generation within the basic blocks from the actual data. The steps in the loop abstraction algorithm are defined below. Each step is accompanied by a narrative description of the performed actions. The detailed description of the algorithm can be found in [18].

a) Identify each control flow statement (simple or compound) which participates in determining a path of a loop.

This step is implemented by identifying the output guards of the basic blocks that are repeatedly activated. The loop abstraction algorithm starts from a loop basic block annotated by *'Loop Label'* during simulation of the program executions (see section 2 for details).

b) Determine the number of exit paths from each control flow statement that determines the loop.

The algorithm conducts a trivial syntactic analysis of the basic block code and determines the number of branches of the block output guards that control output events. For example, there exist *four* outputs controlled by an output guard of basic *block D* of the *Consumer-Producer* example (see **Figure 3.**).

[7] The basic block structure of the xUML programs is explicitly preserved by special words *'state'* and *'endstate'* in the textual representation of xUML programs for the beginning and the end of the basic block respectively. This allows syntactic identification of the code that corresponds to each xUML basic block.

c) Replace each block output guard with a control flow statement with the same set of exit paths where exit path selection is determined by a variable (*path_selection* variable) whose range is defined by the number of outputs (computed in step b) controlled by each block output guard. The values of the *path_selection* variables are non-deterministically chosen. The transformation algorithm is trivial and is performed by copying statements and replacing the conditions of the block output guard by equality comparison of the *path_selection* variable to one value in its range. There are several patterns of the possible configurations of the control tree defined by the block output guard. The transformation algorithm resolves each pattern accordingly such that each transformed output guard truly represents the original program structure. An example of an output guard transformation for the basic *block D* from the *Consumer-Producer* example is given below. The right side represents the original text of the basic block and the left side demonstrates the result of the syntactic transformation.

```
Abstract Basic Block                  |   Concrete Basic Block

path_selection := any(1,2,3);         |if((i<limit_i)&&(j<limit_j)) {
if(path_selection == 1) {             |  Generate e5(PRODUCER,i,j);
  Generate e5(PRODUCER,i,j);          |  Generate e3(CONSUMER); }
  Generate e3(CONSUMER);  }           |else {
else {                                | if((i>=limit_i)&&(j>=limit_j)){
if(path_selection == 2){              |   Generate e3(CONSUMER); }
    Generate e3(CONSUMER);  }         | else {
else {                                | if(i>=limit_i){
  if(path_selection == 3){            |   Generate e5(PRODUCER,i:=0,j);}
     Generate e5(PRODUCER,i:=0,j);}   | else {
      else {                          |   Generate e5(PRODUCER,i,j:=0);}
     Generate e5(PRODUCER,i,j:=0);}   |     }  }
     } }                              |
```

d) Identify all of the variables which depend on the variables which appeared in the control statements that determine the loop. For example, local variables i and j of the *PRODUCER* program used in the *block A* are the dependent variables of the abstracted variables of the *CONSUMER* program's *block D*.

e) Identify all of the control flow statements which are defined over the variables detected in step d. For example, *if* control statements defined over the local variables *i, j* of the basic *Block A* are such control statements.

f) Replace these control flow statements following steps b) and c).

The algorithm terminates when all control flow statements defined over the loop control flow variables and their dependency set are detected.

In addition to the program analysis and transformation, the loop abstraction algorithm automatically creates a file that is used to store the *fairness assumptions* specified as one of the results of the program transformation. Fairness constraints are specified for each value of each *path_selection* variable. For example, during transformation of the *Block D* of the *Consumer-Producer* program, the following set of the fairness constraints is created[8] *Assume Eventually*

[8] The assumptions are encoded in a query language of COSPAN.

path_selection:= 1; *Assume Eventually path_selection*:= 2; *Assume Eventually path_selection* := 3. The file containing the fairness constraints is passed to the model checker and used to specify assumptions (using the *assume-guarantee* features of the model checker, COSPAN) that assure that all outputs defined in the concrete system are explored during the model-checking of the abstract program.

The abstraction is enforced by the syntactic transformation of the basic blocks that are executed in the control loop. The formal definition of the abstract program is $P^a = (X^a, E^a, I^a, B^a)$, where E^a, I^a are defined as for the concrete program, $X^a = X \cup X^{new}$, where X is the set of variables defined in the concrete program and X^{new} is the set of the *path_selection* variables; $B^a = B \setminus B^{loop/depend} \cup B^{new}$, where B, $B^{loop/depend}$ are the sets of concrete basic blocks such that B^{loop} is the loop basic block and B^{depend} is its dependency set (as defined in section 3.1); and B^{new} is the set of the transformed B^{loop} and B^{depend} basic blocks.

4.2 Soundness of the Loop Abstraction

We demonstrate that the loop abstraction is sound with respect to the control flow representation of the concrete program. The soundness result implies that a control specification holds for the original program if it holds for the abstract program.

Let C be an ATS instance associated with the concrete program. Let A be an ATS instance associated with a program that was constructed from C by applying the loop abstraction algorithm.

Theorem 1:

Given a control property φ, the abstract ATS (A) is equivalent with respect to φ to the original ATS (C).

Proof Sketch:

The claim is proved by a trace containment test. We demonstrate that a control trace which conforms to the specification of the control property (see def. 12) of C is contained in the language of control traces, *R.L(A)*.

1. $X^C \subseteq X^A$. This follows the definition of the abstract program (see Section 4.1): (during program transformation new variables, the *path_selection* variables, are added to the program).

2. $E^C = E^A$. This follows the definition of the abstract program (see Section 4.1): (during program transformation *no* new events are added to list of the program events nor are any events of the concrete program are omitted).

3. Generation of events is controlled by the output guards. Call the variables that are used in the output guards of the concrete program, control variables, $X_{control}$. Call the rest of the variables of the concrete program, data variables, X_{data}. Therefore, $X = X_{control} \cup X_{data}$.

The *path_selection* variables are the control variables of the abstract program since they are used in the guard statements to control generation of events. Therefore, from 1 it follows that $X^C_{control} \subseteq X^A_{control}$.

4. Assume that the language of control traces is defined by a set of control traces each of which is initiated by valuation of a different control variable. Let's call these control traces *elementary* control traces.

From 2 and 3 it follows that any *elementary* control trace of C is a subset of $R.L(A)$.

5. From definition of traces (def. 7), every prefix of a trace is a trace. Since the set of initial states is not empty, and the transition relation is serial, every trace can be extended. Therefore, a control property (def. 12) is a specification that is defined over a set of elementary traces. Thus, from 4, any control trace conforming to a control property specified for C is contained in $R.L(A)$.

Therefore, A is equivalent to C with respect to the control property.

It can be shown in the manner above that R.L(C) $\subseteq$ R.L(A) which implies (see def. 11) that C weakly refines control of A and preserves *all* control properties. This means that the same abstraction can be used to check all control properties.

5 Evaluation of the Loop Abstraction Technique

The loop abstraction technique has been evaluated during verification of a NASA Robot Controller System (RCS) formulated as xUML programs. Due to the space limitations we present partial results of the RCS verification and refer the reader to [12, 18, 19] for the detailed description of the RCS and its properties.

Table 1. Verification properties

N	Property	Robotic Description	Formal Description
1	EventuallyAlways(p=1)	Eventually the robot control terminates	Eventually permanently p=1
2	NeverUntil(r=1,fk=1)	No command to move the robot arm is scheduled before an initial position of the arm is computed	It is never the case that r=1 holds until fk=1 holds

Sample properties (both safety and liveness) are given in **Table 1**. The properties are encoded in a query language of COSPAN and refer to variables of the RCS xUML programs. For example, declaration p refers to the *abort_var* variable of the *Controller* RCS xUML program, *fk* refers to the *forward_kinematics* variable of the *EndEffector* RCS xUML program, and r refers to the *ee_reference* variable of the *Trajectory* RCS xUML program.

We considered several variants of the RCS of different complexity defined by the number of joints i of a robot arm (i defines degrees of freedom (DOF) of the robot arm). We used two types of programs to check the properties. The first type is the complete (concrete) program. The second type is the abstract version of the concrete program to which the loop abstraction method has been applied.

Table 2. Comparison of Verification of the Concrete and Abstract Robotic Systems

i	P1: Concrete (states/m:s/MB)	P1: Abstract (states/m:s/MB)	P2: Concrete (states/m:s/MB)	P2: Abstract (states/m:s/MB)
2	2.2e+12/350:4/735	26K/0:28/4.03	2.3e+11/344:4/713	17K/0:17/3.38
3	3e+18/415:4/1,246	63K/3:10/4.9	2e+17/410:3/1,190	63K/3:10/4.9
4	6e+23/592:4/1,802	145K/11:28/8.4	6e+24/662:3/2,190	116K/7:03/7.1
5	M/T exhaustion	688K/28:10/23.9	M/T exhaustion	554K/13:40/19.1
6	M/T exhaustion	1.1M/42:17/96.5	M/T exhaustion	715K/33:17/36.2

Table 2 compares the run-time and memory usage for properties from Table 1. The results are given for the concrete and the abstract RCS with a total number of 7 xUML programs. They exclude the i programs corresponding to the number of instances of the *Joint* object. Each entry in the table has the form $x/y/z$ where x is the number of the states reached, y is the run-time in cpu minutes and seconds (m:s) and z is the memory usage in Mbytes (MB). The results of the verification demonstrate significant reduction in both time and space for the abstract program compared to the concrete program. The results are given for the explicit state space exploration experiments and demonstrate that the reduction becomes more pronounced for larger values of i. Verification for the robot configurations consisting more than 4 joints (4 DOF) could not be completed for the concrete program due to the memory/time exhaustion (denoted as *M/T exhaustion* in Table 2), but COSPAN *succeeded for the abstracted model*. It is notable that application of the state space reduction techniques such as BDD-based verification and partial order reduction to verification of the concrete program also resulted in the state space explosion for programs implementing control of robots with more than 4 DOF. Our efforts on the predicate abstraction of the concrete program also failed. Specifically, the predicate abstraction technique supported by COSPAN [15] did not succeed due to the memory exhaustion during computation of the abstraction predicates. The boolean abstraction approach [1] resulted in over-approximation of the program executions that led to the state space explosion during verification of the abstracted program even for programs with less than 5 DOF. Thus, the loop abstraction technique became the *only* approach that enabled verification of robot configurations higher than 4 DOF.

6 Conclusions, Future and Related Work

Conclusions. The paper presented an approach for *practical* model checking of large-scale software. A loop abstraction technique has been defined and implemented in the context of the integrated design and model checking software development. The abstraction algorithm is computationally simple and requires storage linear in the size of the program since it is a source to source transformation. It proved to be highly effective in state space reduction for the test-case control-intensive program, the large-scale robot controller system. Most importantly, the loop abstraction enabled completion of model checking for realistic robot configurations where all other approaches, including predicate abstraction [1, 15], failed.

It would be expected that a selective and limited scope abstraction such as the loop abstraction would introduce fewer unrealistic behaviors into the abstract program than more comprehensive abstractions. This proved to be the case for the robot control system. Only a few refinements were needed. These were identified as false negatives in model checking the abstract program and were manually implemented. Propagation of the abstraction across basic blocks, planned as a future work, will further reduce the number of unrealistic behaviors introduced by the abstraction and hence the requirement for refinement.

A limitation of the loop abstraction is that it can only be applied when the properties to be model checked are control properties. Control properties are, however, typically the safety-critical properties of control systems.

Related Work. Loop abstraction is similar to predicate abstraction in that it requires specification of an abstraction function as predicates over concrete data. Loop abstraction differs from predicate abstraction in that it does not require computation of the abstraction predicates. Instead it operates on the conditional predicates which implement program control. The result of the loop abstraction is the construction of a control skeleton which makes our work similar to construction of boolean programs as defined in [1]. However, our work is different from [1] in that it is concerned with the abstraction of only the loops. Loop abstraction introduces a *limited number* of unrealistic behaviors compared to [1] and also preserves some original data valuations compared to the complete data abstraction provided by predicate abstraction methods. Loop abstraction can be a useful complement to predicate abstraction. It abstracts control while predicate abstraction abstracts statements not effected by the loop abstraction.

The implementation of the loop abstraction algorithm is similar to [15] in that the loop abstraction algorithm does not construct the explicit state graph of either the original or of the abstract program. Instead a syntactic analysis of the original program is used to produce an abstract program. However, our approach is different from other abstraction algorithms dealing with the source code in that the abstraction is applied to a design-level specification (xUML programs). To our knowledge, there has been no previous reports on data abstraction algorithms specifically targeting design level specifications.

The work presented in this paper is also related to path coverage (also known as predicate coverage) testing [2, 3]. Path coverage reports whether each of the possible paths in each function of the program has been followed. (A path in testing is a unique sequence of branches from a function entry to exit). Loop abstraction provides complete coverage of all possible execution paths within a loop. One of the major obstacles to successful path coverage is looping during program execution. Since loops may contain an unbounded number of paths, path coverage only considers a limited number of looping possibilities. Our method solves this problem. Path coverage has the problem that many potential paths are impossible to reach because of data relationship constraints. Loop abstraction algorithm solves this problem by adding fairness constraints to force exploration of all abstracted paths.

References

1. T. Ball, R. Majumdar, T. Millstein and S. Rajamani, Automatic Predicate Abstraction of C Programs, *In Proc. of PLDI 2001, SIGPLAN Notices*, Vol. 39 (2001)
2. B. Beizer, *Software Testing Techniques*, New York: Van Nostrand Reinold, (1990)
3. J. J. Chilenski and S. P. Miller, *Applicability of modified conditional coverage to software testing*, Software Engineering Journal, (1994) 193 - 200
4. E. M. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *In Proceedings POPL 92: Principles of Programming Languages*, (1992) 343 - 354
5. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for the static analysis of programs by construction of approximation of fixpoints. *In Proc. of POPL'77*, (1977) 238 - 252
6. D. Dams, O. Grumberg, and R. Gerth. Abstract interpretation of reactive systems: abstractions preserving ACTL*, ECTL*, and CTL*. *In Proceedings of PROCOMET 94: Programming Concepts, Methods, and Calculi*, (1994) 561-581
7. S. Graf and H. Saidi, Construction of abstract state graphs with PVS. *In Proceedings of CAV 1997*, LNCS 1254 (1997) 72 - 83
8. R. Hardin, Z. Har'EL, and R. P. Kurshan, COSPAN, *In Proceedings of CAV 1996*, LNCS 1102, (1996) 423 - 427
9. M. S. Hecht, *Flow Analysis of Computer Programs*, NY: Elsevier-N. Holland (1977)
10. J. Holzmann, *Design and Validation of Computer Protocols*, Pr. Hall, NJ (1991)
11. Kennedy Carter Inc., www.kc.com
12. Kapoor, C., and Tesar, D.: A Reusable Operational Software Architecture for Advanced Robotics (OSCAR), The University of Texas at Austin, DOE Grant No. DE-FG01 94EW37966 (1998)
13. Kurshan, R., *Computer-Aided Verification of Coordinating Processes - The Automata Theoretic Approach*, Princeton University Press, Princeton, NJ (1994)
14. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, Vol. 6(1), (1995) 11-44
15. K. S. Namjoshi and R. P. Kurshan, Syntactic Program Transformations for Automatic Abstraction, *In Proc. of CAV'00*, LNCS 1855, (2000), 435-449
16. Y. Kesten and A. Pnueli, *Control and Data Abstraction: Cornerstones of the Practical Formal Verification*, Software Tools and Technology Transfer, Vol. 2(4) (2000)
17. SES Inc., *ObjectBench Technical Reference*, SES Inc. (1998)
18. N. Sharygina, *Model Checking of Software Control Systems*, Ph.D. Dissertation, The University of Texas at Austin (2002)
19. N. Sharygina, J. C. Browne and R. Kurshan, A Formal Object-Oriented Analysis for Software Reliability: Design for Verification, *In Proc. of FASE'01*, LNCS 2029, (2001), 318-332
20. S. Shlaer, S. Mellor, *Object Lifecycles: Modeling the World in States*, Pr. Hall (1992)
21. L. Starr, *Executable UML: The Models that Are the Code*, M. Integration, LLC (2001)
22. F. Xie, V. Levin, and J. C. Browne, Model Checking of an Executable Subset of UML, *In Proceedings of ASE2001: Automated Software Engineering* (2001)