Loop Abstraction for Model Checking Large-Scale Software

Natasha Sharygina and James C. Browne

The University of Texas at Austin,
Austin, TX, USA 78712
natali,browne@cs.utexas.edu

Abstract. This paper reports on the design, implementation and evaluation of a data abstraction algorithm which is effective in reducing the complexity of model-checking for control properties of large-scale programs. The reduction technique performs a transformation of a "concrete", possibly infinite state, program by means of a *syntactic* program transformation that results in an "abstract" program that when model-checked provides complete but minimal coverage of cyclic execution paths. 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 the program. The loop abstraction has been implemented in the integrated xUML design, testing and formal verification software development environment. We use as a case study a NASA robot control system and report on substantial reduction in both time and space for the abstract model compared to the concrete model.

Keywords: Abstract Model Checking, Software Verification, Integration of Software Design, Testing and Verification

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 detect 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 for minimizing the contribution of the loop executions to program state space. A rationale for the effectiveness of the loop abstraction is given following.

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 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 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 absence of deadlock of the program execution or guarantee that a functionally unsafe state will not be reached) are dependent only on the static 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 static 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 that the concrete programs from which they are derived.

The abstraction algorithm is computationally simple and requires storage only linear in the size of the program since it is a source to source transformation based on static analysis of the program.

The steps in the algorithm are:

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

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

c) Replace each control flow statement with a simple control flow statement with the same set of exit paths where exit path selection is determined by a random variable which ranges over the number of exit paths.

d) Identify all of the variables which depend on the variables which appeared in the control statements that determine the loop.

e) Identify all of the control flow statements which depend on the abstracted loop control flow variables.

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

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 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.

Additionally, since the loop abstraction is applied to the source code, it has the following advantages over abstraction methods that construct explicit transition graph.

- it is independent of the model checker or the verification algorithm. Thus other state space reduction techniques, such as symbolic model-checking and partial order reduction, can be applied to the abstract program.

- it can be applied to any representation of the program including the design level specifications.

The loop abstraction algorithm has been implemented in a front-end of the FormalCheck/COSPAN model checking tool 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. Moreover, it enabled model-checking of control properties for 5 and 6 joint robot arms which had previously been intractable with available computational resources.

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.

Contents of Paper. Sections 2 defines syntax and semantics of the control software systems. Section 3 defines the loop abstraction. Section 4 describes an implementation of the loop abstraction algorithm in the framework of integrated software design and model checking. The effectiveness of loop abstraction is demonstrated in Section 5 that shows the verification results of the NASA robot controller system. Section 6 concludes the paper and positions the loop abstraction with respect to the existing abstraction techniques.

2 Background

2.1 Program Syntax

Control software systems are often constructed as *compositions of sequential programs* which interact through sending of messages or events.

Definition 1 [**Sequential Program**]**:** *A sequential* program *is defined as follows:*

$$SeqProc \rightarrow Proc;\ terminate,$$

where *Proc* is defined by commands[1]:
simple commands:

$$x := exp \mid x := any\{\ exp_1,...,exp_n\ \}\ \mid$$

compound commands:

$$Proc1,Proc2 \mid if\ B\ then\ Proc1\ else\ Proc2\ fi \mid while\ B\ do\ Proc1\ od \mid$$

communication commands:

$$'Generate\ e(ID,exp)' \mid 'Receive\ e(ID,x)'\ .$$

In the above definitions x is a program variable, exp_i are expressions over program variables, and B is a boolean condition, e is the name of the event, ID is the name of the event destination program. The statement $x := any\{\ exp_1,...,exp_n\ \}$ is a *non-deterministic* assignment, after which x will contain the value of one of the expressions $exp_1,...,\ exp_n$.

We can, without loss of generality, assume that each program is comprised of basic blocks [9].

Definition 2 [Basic Block]: *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 3 [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 1 illustrates the concept of the basic block. The control flow graph (at the command level) illustrates the control flow paths that determine the outputs of the basic block.

Definition 4 [System]: *A* system *is a parallel composition of sequential programs. Each program has its own read-shared local variables and events.* In general terms a system, S, is defined as a set of variables, X, and a set of events, E, an initial condition, I, a set of basic blocks, B, that contain commands that modify the program variables, and send and receive events, $S = (X,E,I,B)$.

[1] For the complete list of the commands see [19]

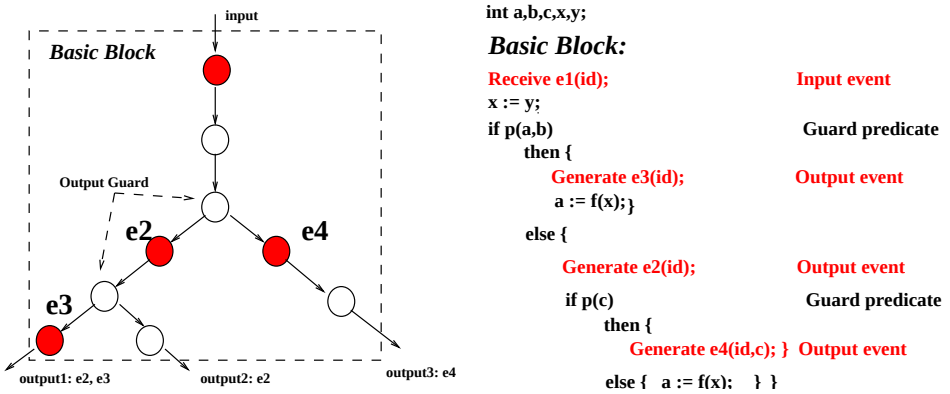

Fig. 1. Demonstration of a Basic Block Concept

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.

This works exploits the *atomicity* of the program executions based on the basic block view of the program structure.

Definition 5 [**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.

Definition 6 [**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).*

Each loop is guarded by a set of the basic block output guards of the loop basic block and their dependence set. Let us call the variables of the basic block output guards that define the loop, *loop variables*.

2.2 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 7 [**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 8 [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 9 [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 10 [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 11 [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 conditions** that determines which traces of the model are confronted with the specification during model-checking. A fairness condition is defined as a boolean combination of basic fairness conditions "infinitely often p" where p is a set of state pairs. The *trace is fair* if the fairness condition is true in infinitely many states along the trace.

Definition 12 [Refinement]: Let A and C be two instances of the ATS defined preceding. Let L(A) and L(C) be the language of all traces from execution of A and C.

$$\text{If } X^C \subseteq X^A, \text{ and L(C)} \subseteq \text{L(A) then C } \textit{weakly refines} \text{ A, C} \leq \text{A.}$$

Definition 13 [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.

$$\text{If } X^C \subseteq X^A \text{ and R.L(C)} \subseteq \text{R.L(A) then C } \textit{weakly refines control} \text{ of A.}$$

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 14 [Control Property]: *A* control property *is a temporal logic specification defined over states that input events.*

3 Loop Abstraction

We define a loop abstraction technique that maps all of the traversals of a loop in the program control flow graph with different values for the loop variables to traversals with values of newly introduced variables whose range is the number of the basic block outputs. The values of the new variables are non-deterministically chosen subject to fairness constraints. The loop abstraction is the syntactic program transformation that results in a reduced ATS that provides complete but minimal coverage of the program executions and, that, thus, can be practically model-checked.

We present the abstraction *informally* by specifying the loop abstraction algorithm. We demonstrate the soundness of the abstraction *formally* by presenting a proof of correctness of the loop abstraction (see Appendix B).

3.1 Loop Abstraction Algorithm[2]

We use the basic block control flow graph of the program to derive information that we need to implement the loop abstraction. We compute a number of outputs for each basic block that is executed in the control loop and use the computed data to abstract the generation of events within the basic blocks from the actual data. The abstraction is enforced by the syntactic transformation of the basic blocks.

We first present components of the loop abstraction algorithm: an algorithm for computation of a number of outputs controlled by a basic block output guard and a basic block output guard transformation procedure. We conclude by presenting an algorithm for the loop abstraction.

Basic Block Output Range Computation. The range computation algorithm (*compute_range*) performs syntactic analysis of the block output guard by parsing its text and searching for the *Generate* and *if, while* keywords. A sketch of the algorithm is given in **Figure 2**. The number of branches of the block output guard is counted and stored in the *range* variable. Since each branch defines a basic block output (see def. 3) then the *range* variable defines the number of possible outputs controlled by a basic block.

The *compute_range* program maintains two variables which are used to store information required during the analysis of the programs of the guards, *branch* and *found*, declared as integer and boolean respectively. Initially (step 1) both variables are set to zero.

Guard Transformation. In the abstract program, the block output guards are substituted with multi-way selector expressions, *Path Selectors*, each of which non-deterministically selects the outputs to be generated during an execution of the basic blocks. Each *Path Selector* is defined over a single variable, a *path_selection* variable, with a range defined by the number of the outputs controlled by the corresponding

[2] The loop abstraction technique is defined and implemented for xUML software systems. xUML is an instantiation of the programming model defined in section 2. xUML notation supports separation of data and control. This separation enables syntactic identification of the program basic blocks that are used to determine the program control structure. Specification and an example of the xUML programs can be found in **Appendix A**.

```
int compute_range()  {
Step 1.   If (branch==0) {
              Goto Step 2 and parse commands of a positive test program
          Else Goto Step 2 and parse commands of a negative test program }
Step 2.   Until (the end of the body of the conditional statement is reached {
              If ( Generate keyword found AND found !=1) {
                   range++; found:=1;}
              If ( if or while keyword found ){
                   Goto Step 1; found := 0; } }
Step 3.   branch++;
              If (branch == 2) Goto Step 4
              Else Goto Step 1
Step 4.   return range; }
```

Fig. 2. A Sketch of the Output Range Computation Algorithm

block output guard. Each output controlled by the *Path Selector* is selected by a single value of the *path_selection* variable. Subject to the fairness constraints specified for all values for each *path_selection* variable, the global state transition graph of the abstract program will have all of the event sequences and thus interleavings of basic block executions as the global state transition graph of the concrete program. The transformation algorithm is trivial and is performed for each basic block by copying commands 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 *Path Selection* expression truly represents the original program structure. An example of an output guard transformation for a sample basic block 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.

```
int a,b,c,x;

Abstract Basic Block                    |   Concrete Basic Block

Receive e1(id);                         |    Receive e1(id);
//actions omitted                       |    //actions omitted
path_selection := any(1,2,3,4);         |   if(p(a,x)) {
if(path_selection == 1) {               |       Generate e5(id);
  Generate e5(id);                      |       Generate e3(id,x); }
  Generate e3(id,x);  }                 |   else {
else {                                  |       if(p(a,c))
  if(path_selection == 2) {             |          Generate e3(id2,a,c);  }
     Generate e3(id,a,c);  }            |       else {
  else {                                |          if(p(c))
     if(path_selection == 3) {          |             Generate e5(id,c);
       Generate e5(id,c);}              |           if(p(b))
     if(path_selection == 4)  {         |             Generate e5(id,b));
       Generate e5(id,b);}              |            } }
     } }
```

The Loop Abstraction Algorithm. The loop abstraction algorithm performs syntactic transformation of the program text by *transformation of basic blocks*. The abstraction algorithm starts from the transformation of a *loop basic block* by substituting its output guards with the *Path Selector* expressions[3]. The abstraction of a concrete set of output guards may introduce some loss of information caused by the non-deterministic choice over the set of the *path_selection* variables. To compensate the imprecision, abstraction of the loop control flow statements that depend on the loop control flow variables is performed. A sketch of the algorithm is presented in **Figure 3**.

Step 1. Detect *loop labels*
Step 2. Analyze and Transform a Basic Block, $B^{loop/depend}$:
While end of basic block is not reached {
(a) *current_command* ← command
(b) **If** *current_command* is a guard {
If guard is an output guard {
- *abstract_guard*++
- *flow_var_storage* ← names of the *loop variables*
- *range_size* ← *compute_range*
- *transform_guard*(*range_size*,*abstract_guard*)
- *fairness.txt* ← assumptions(*range_size*,*abstract_guard*) } } }
(c) Move to the next command }
// A new program is formed by substitution of $B^{loop/depend}$
with the transformed basic block
Step 3. Detect depend variables:
While depend variables found {
flow_var_storage ← names of the *depend* variables }
Step 4. Detect depend guard:
(a) **While** guard found {
(b) **If** guard is an output guard {
(c) **If** guard is the depend guard {
break; **Goto** Step 2 } } }
Step 5. An abstract program is formed as: $P^a = (X^a, E^a, I^a, B^a)$,
where E^a,I^a are defined as for the concrete program,
$X^a = \mathrm{X} \cup X^{new}$, X is the set of the concrete variables and
X^{new} is the set of the *selection* variables.
$B^a = \mathrm{B} \setminus B^{loop/depend} \cup B^{new}$, where
B, $B^{loop/depend}$ are the sets of concrete basic blocks,
and B^{new} is the set of transformed basic blocks.

Fig. 3. A Sketch of the Loop Abstraction Algorithm

The loop abstraction algorithm maintains four variables which are used to store information required during the analysis and transformation of the program, *range_size*, *abstract_guards*, *current_command*, and *flow_var_storage* declared as integers, char and an array of char type respectively. It also creates a file, *fairness.txt* that is

[3] A loop basic block is identified and labeled by a *'Loop Label'* during simulation of the program executions (see section 4).

used to store the fairness assumptions specified as one of the results of the program transformation.

The algorithm proceeds as follows:

Step 1: The loop markers are identified (the program looks for a keyword 'Loop Label').

Step 2: The algorithm iteratively analyzes and transforms basic blocks that define control flow within a loop[4].

At each iteration the algorithm parses the text of the basic block command after command while performing the following series of actions:

a) Placement of the first command found in the text of the basic block into *current_command*;

b) Testing if the *current_command* is a guard (i.e test against the *if, while* keywords). For the negative test the program proceeds to step 2c, otherwise the following actions are performed:

If the guard is the *output guard* (i.e. if the body of the guard includes the events generation commands (denoted by the *'Generate'* keyword)) then

- The *abstract_guard* variable that is used to store a number of transformed guards is updated.

- The names of the *loop variables* of the output guards are passed to the *flow_var_storage* variable.

- The *compute_range* program is invoked to count the number of outputs controlled by the guard. The result is stored in the *range_size* variable.

- The *transformation* program uses the current values of the *abstract_guard* and *range_size* variables to transform the guard following the procedure discussed in the guard transformation section.

- The *fairness.txt* file is updated with new fairness constraints. The fairness constraints are defined following the scheme:

For (int $i := 0$, int $j :=$ *abstract_guard* - 1; $i \leq$ range_size; i++) {
Create a line: *AssumeEventually*[5] *path_selection*[j] := i }[6].

The file containing the fairness constraints is later used to specify assumptions that assure that all outputs defined in the concrete system are explored during the model-checking of the abstract program.

c) If the end of the basic block is not reached then the next command of a basic block is analyzed (if the previous command was a guard that was transformed, then the program starts from the command that follows the new structure).

[4] The atomic structure of the xUML programs is explicitly preserved by special words *'state'* and *'endstate'* 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.

[5] The assumptions are encoded in the query language of the COSPAN model-checker, a part of the loop abstraction implementation environment.

[6] For example, if the first output guard found during the program analysis (*abstract_guard* == 1) controls four outputs (*range_size* == 4), then the following set of the fairness constraints is created: (Assume Eventually *path_selection*[0] := 1; Assume Eventually *path_selection*[0] := 2; Assume Eventually *path_selection*[0] := 3; Assume Eventually *path_selection*[0] := 4).

Step 3: Dependency analysis between the program variables and the variables stored in the *flow_var_storage* is performed. The names of the dependent variables are passed to the *flow_var_storage* variable.

Step 4: The new program is searched for *depend guards* (i.e. conditional statements 'if-then-else/while', which operate on the variables that are included in the *flow_var_storage* array). If a depend guard is found then program proceeds to step 2, otherwise it proceeds to Step 4.

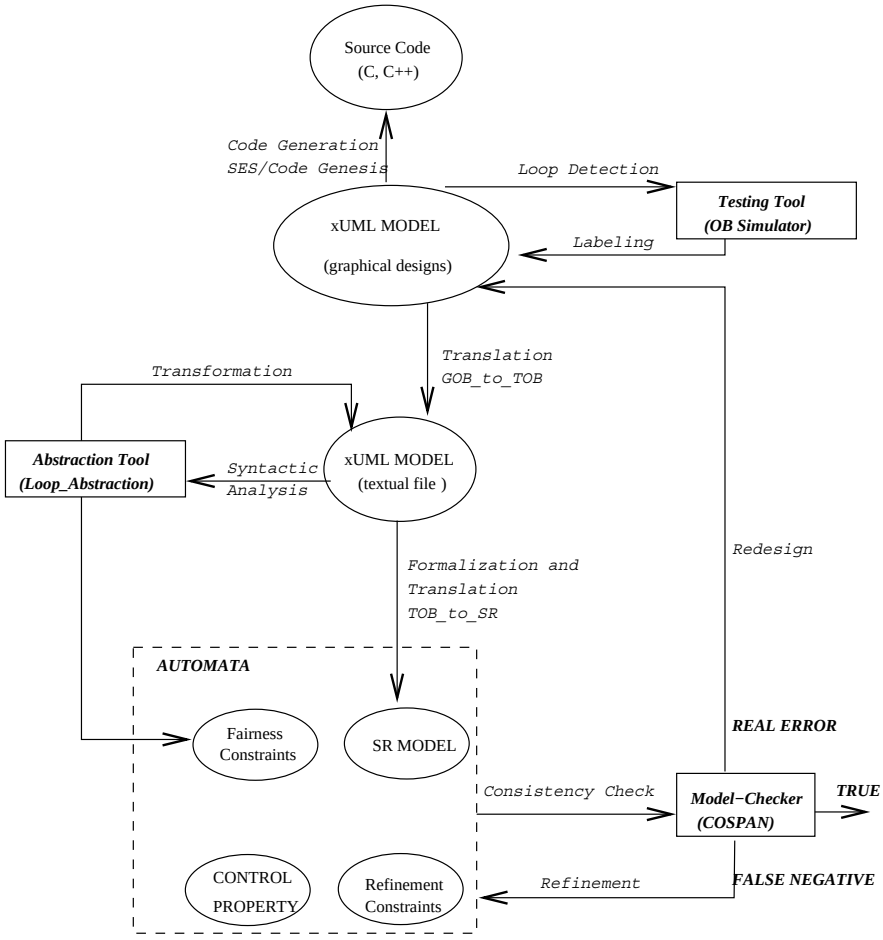

Fig. 4. The Loop Abstraction Procedure For the Integrated Design and Model-Checking Software Development Environment

Step 5: Program transformation finishes by inserting into the declaration part of the program text a line that declares an array of variables *path_selection*[*abstract_guard*] of integer type.

The abstract program is the source-to-source transformation of the program during which an array of *path_selection* variables is added to the program specification and all basic blocks that determine the control flow of the program loops are substituted with the *Path Selector*-controlled basic blocks. See **Figure 3**, Step 5 for formal definition of the abstract program.

4 Implementation of the Loop Abstraction

The loop abstraction has been implemented in the software development framework that integrates xUML modeling, testing and automata-based model-checking. We refer the reader to [22], [25] for the detailed description of the integrated environment. The steps of the loop abstraction procedure as they are implemented in the integrated design and verification environment are captured in **Figure 4**. The abstraction procedure operations are supported by the following tools (each tool is represented with respect to the actions it performs in the loop abstraction procedure):

1. *The xUML graphical specification and validation environment as it is implemented in the commercial tool,* SES/ObjectBench (OB) [19]:

- a *loop* in the execution behavior of the xUML programs are *detected* using the discrete event simulator by traversing possible *event sequences* which can arise from the execution of interacting xUML state machines;

- a *basic block* that are identified to be repeatedly activated is manually annotated with a *'Loop Label'* in the xUML specification environment.

2. *The* loop_abstraction *program*:

- the labeled xUML state machines are *syntactically analyzed* and *transformed* into the abstract xUML state machines using the loop abstraction algorithm.

- a set of the *fairness constraints* is *generated*. The list of the generated fairness constraints is passed as an input to the model-checker.

3. *The automata-based model-checking tool,* COSPAN[8]:

- a *consistency check* is performed over the abstract SR model (SR is an input language of COSPAN) automatically derived[7] from the abstract xUML program with respect to the the specified control property, the fairness constraints and the approximation restrictions. The following features provided by COSPAN are used:

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

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

[7] The abstraction procedure uses a translator [25] 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.

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 models. Description of the the RCS *Kinematics* component, which verification results are discussed here, can be found in **Appendix A** (for detailed information see [12, 21, 22]).

Table 1. Verification properties

N	Property	Robotic Description	Formal Description
1	EventuallyAlways(p=1)	Eventually the robot control terminates	Eventually permanently p=1
2	AfterAlwaysUntil(q=5, r=1, p=1)	When *EE* reaches a 'NotValidPosition' state program terminates prior to a new *EE* move	At any point in the execution if q=5 than it is followed by r=1 until p=1
3	Always(r=1 $\rightarrow$ u=0)	If the *EE* is the "FollowingTrajectory" state than the *Arm* is in the "Valid" state	At any time during execution of the program when r=1 than u=0

Sample properties (both safety and liveness) are given in **Table 1**. The properties are encoded in a query language of COSPAN. Since it is easier to reason about the program control flow in terms of the locations in the program execution rather than in terms of events, we specify the *control properties* in terms of the states defined by the *labeling variables* (defined in Appendix A) in the xUML system[8]. For example, the labeling variable *ee_status* of the *EndEffector* program can change its values within the following range { *'Idle', 'FollowingTrajectory', 'CheckingConstraints', 'ValidPosition', 'NotValidPosition', 'InitialPositioning'* } (see Figure 5, Appendix A), which for the purpose of model checking was encoded during translation into the following range of integers {0,1,2,3,4,5}.

The following declarations are used in Table 1: p - declares the *global_status* variable of the *Global* program; q - declares the *ee_status* variable of the *EE* program; u - declares the *arm_status* variable of the *Arm* program.

We considered several variants of the RCS of different complexity defined by the number of joints i of a robot arm. We used two models to check the properties. The first model is the complete (concrete) structure of the robot arm. The second model is the abstract version of the concrete model to which the loop abstraction method has been applied.

Table 2 compares the run-time and memory usage for the first two properties from Table 1. The results are given for the concrete and the abstract RCS with a total number of 7 xUML programs excluding 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 seconds

[8] The labeling variables values are preserved by the loop abstraction since they do not depend on any program variables but the fact that an event arrives to a basic block.

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

i	P1: Concrete (states/min:sec/MB)	P1: Abstract (states/min:sec/MB)	P2: Concrete (states/min:sec/MB)	P2: Abstract (states/min:sec/MB)
2	4.02M/212:35/211	26K/0:28/4.03	1.97M/83:42/145	17K/0:17/3.38
3	5.54M/301:50/289	63K/3:10/4.9	3.48M/253:42/246	45K/2:40/2.2
4	7.34M/550:10/474	145K/11:28/8.4	5.89M/367:38/302	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

and z is the memory usage in Mbytes. The results of the verification demonstrate significant reduction in both time and space for the abstract model compared to the concrete model. The reduction becomes more pronounced for larger values of i. Verification for the robot configurations consisting more than 4 joints could not be completed for the concrete model due to the memory/time exhaustion (denoted as *M/T exhaustion* in Table 2), but COSPAN succeeded for the abstracted model.

6 Conclusions and Future Work

Conclusions. We defined a loop abstraction technique that is computationally simple and requires storage only linear in the size of the program since it is a source to source transformation based on static analysis of the program. 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 seems probable that it will be equally effective on other cycle intensive programs.

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.

The 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.

Future Work. The immediate future research includes: automation of the refinement process, application to other cyclic programs including some classical test cases such as the dining philosophers problem and extension to programming systems other than xUML. Propagation of the abstraction across basic blocks will further reduce the number of unrealistic behaviors which are introduced by the abstraction and further reduce the requirement for refinement. Longer term research is a search for other widely applicable and effective selective abstractions.

7 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. We are planning to evaluate the loop abstraction in combination with predicate abstraction as a part of a project that develops a prototype automatic predicate abstraction tool [15].

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 technique solves this problem by adding fairness constraints to force exploration of all abstracted paths.

Acknowledgments. We thank Bob Kurshan, Allen Emerson, Kedar Namjoshi and Nina Amla for their helpful comments. This research was supported in part by the TARP program 003658-0508-1999, by Bell Laboratories Lucent Technologies, and by the University of Texas at Austin Robotics Research Group.

References

1. T. Ball, R. Majumdar, T. Millsteain and S. Rajamani, Automatic Predicate Abstraction of C Programs, *In Proceedings 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 Proceedings of POPL 77: Principles of Programming Languages*, (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-North Holland (1977)
10. J. Holzmann, *Design and Validation of Computer Protocols*, Prentice 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, Report to DOE, Grant No. DE-FG01 94EW37966 and NASA Grant No. NAG 9-809 (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 Proceedings of CAV 2000: Computer Aided Verification*, 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) 328 - 342
17. ProjectTechnologies Inc., www.projtech.com
18. H. Saidi, Modular and Incremental Analysis of Concurrent Software Systems, *In Proceedings of ASE 1999*, ACM Press (2000) 92 - 101
19. SES Inc., *ObjectBench Technical Reference*, SES Inc. (1998)
20. SES Inc., *CodeGenesis User Reference*, SES Inc. (1998)
21. N. Sharygina, and D. Peled, A Combined Testing and Verification Approach for Software Reliability, *In Proc. of FME2001: Formal Methods Europe*, LNCS 2021, (2001) 611-628
22. N. Sharygina, J. C. Browne and R. Kurshan, A Formal Object-Oriented Analysis for Software Reliability: Design for Verification, *In Proceedings of ETAPS2001(FASE): Fundamental Approaches to Software Engineering*, LNCS 2029, (2001), 318-332
23. Shlaer, S., and Mellor, S., *Object Lifecycles: Modeling the World in States*, Prentice-Hall, NJ (1992)
24. L. Starr, *Executable UML: The Models that Are the Code*, Model Integration, LLC (2001)
25. F. Xie, V. Levin, and J. C. Browne, Model Checking of an Executable Subset of UML, *In Proceedings of ASE2001: Automated Software Engineering* (2001)

Appendix A. xUML Programming Paradigm

xUML [23] is an instantiation of the programming model described in section 2.1. 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, 19].

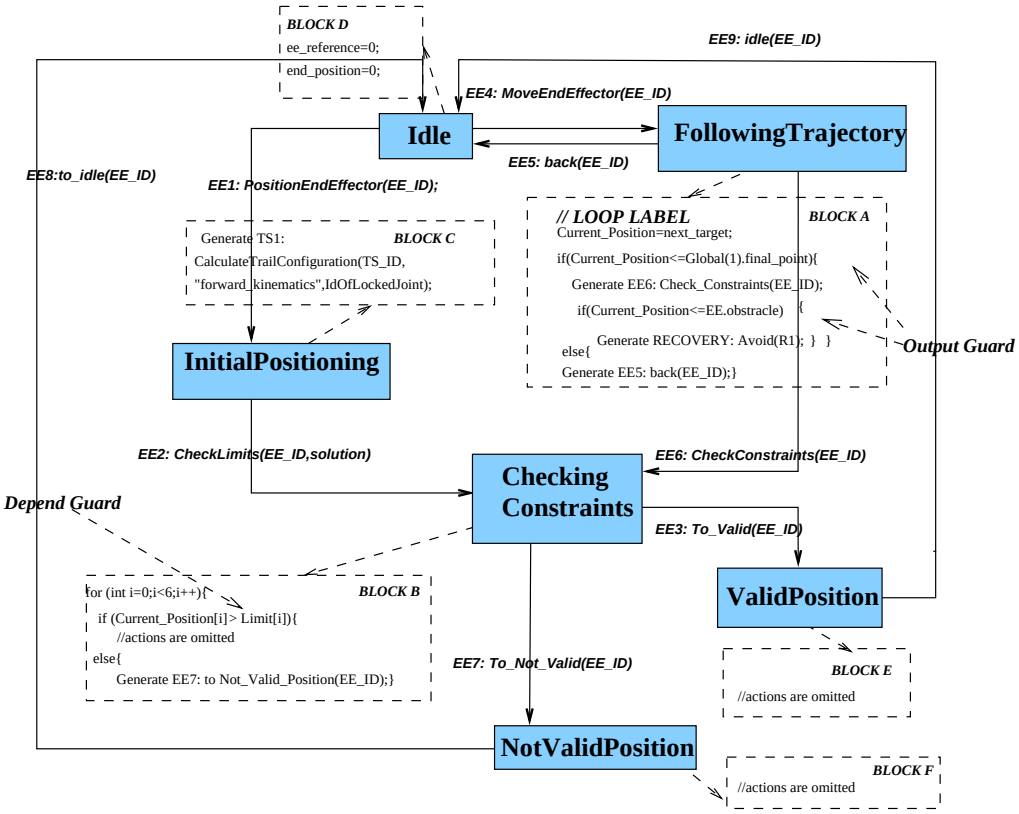

Fig. 5. xUML model of the End_Effector program of the Robot System.

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 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. Each state machine has a state *labeling variable* which is updated immediately following

receipt of an event. The presence of the labeling variable allows reasoning about the control flow in terms of locations in the program execution rather than in terms of events.

Each state machine corresponds to a sequential program of the programming model defined and described in Section 3. Each action corresponds to a basic block of a sequential program. 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. The semantic model for xUML is *identical* to the semantic for the programming model defined and described in Section 3.

An example of the xUML system is given using a *Kinematics* component of the test-bed robot controller system (RCS) [22, 21]. The *Kinematics* component is modeled by the xUML state machines, representing behavioral specifications of the *Arm*, *Joint*, *End_Effector*, *Checker*, *Recovery*, *Trial_Configuration*, *Global_ Representation* xUML objects. **Figure 5** shows the state machine of the *End_Effector* program as an example of an xUML behavioral specification.

The state machine is represented as a collection of basic block that are activated by events. For example, an *Block A* can be activated by an input event *EE4* and labeled by the update of a variable *ee_status := FollowingTrajectory* (in the example, the label variables update commands are implicitly implemented by the xUML graphical development environment.) The activation of the basic block is followed by the execution of local commands and generation of output events *Generate EE6: CheckConstraints(EE_ID), Generate EE5: back(EE_ID), and Generate RECOVERY: Avoid(R1).*

The *Kinematics* RCS component implements the following algorithms:

- *Robot Control Algorithm.* Given a target position of the last joint of the robot arm (end-effector), every joint calculates its target angle position. If each target angle position satisfies the physical constraint imposed on the joint, the arm proceeds to the target position; otherwise, fault recovery is called.

- *Fault Recovery Algorithm.* The position of the joint that violates the physical constraints is set to the specified limit while the other joints recalculate their target angle positions.

- *Obstacle Avoidance Algorithm.* If the robot arm encounters an undesired position (an obstacle in the robot workspace), a new position around an obstacle is searched by the robot arm. If a new position of the arm is found and joint target angles are identified, the robot arm proceeds to the next target position, otherwise robot control terminates.

Appendix B. 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. 14) 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 Figure 3): (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 Figure 3): (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. 9), 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. 14) 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. 13) 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.