Data Abstraction for Cycle Intensive Programs

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 cycle intensive 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 cycle 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: Model Checking Software, Data Abstraction, Integrated Software Design, Testing and Verification

1 Introduction

It is widely believed that effective model-checking of software systems could produce major enhancement in software reliability and robustness. But the effectiveness of model-checking of software systems is severely constrained by the state space explosion problem. One principal method in state space reduction of software systems is abstraction. Abstraction techniques reduce the program state space by mapping the set of data states of the actual system to an abstract set of states that preserve the behaviors of the actual system. Abstraction is widely used and it has been explored by a number of researchers [4, 6, 13, 14, 16]. Abstractions introduce unrealistic behaviors into the specification of the programs. Intelligent refinements of the abstracted programs are required to avoid false negative results from model-checking.

This paper reports on the design, implementation and evaluation of a data abstraction algorithm that *automates* an abstraction process of *cycle* intensive systems. The algorithm given here performs a transformation of a "concrete", possibly infinite state, program by means of a *syntactic* program transformation that results in an "abstract" program which when model-checked provides complete but minimal coverage of program execution paths. Given a control structure for a program, the cycle abstraction algorithm iteratively computes the number of possible outputs of

each control flow statement of the program that effects the cycle control flow. The conditional expressions of each cycle control flow statement are replaced with an expression over a single variable with a value non-deterministically chosen in the range (1,number of control outputs of cycle control flow statement). This is achieved by a syntactic analysis that does not construct the explicit transition graph either of the original or of the abstract program, each of which may be too large to compute. Generation of all original control flows is assured by fairness constraints specified as assumptions on the values of the cycle flow statement variables of the abstract program.

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.

The cycle abstraction algorithm has several advantages:

- it is a *selective* and targeted abstraction which introduces few unrealistic behaviors requiring refinement.

- it applies at the design level for software systems.

- 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. This enables integration with existing tools at a low cost.

- 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 cycle abstraction algorithm 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.

Contents of Paper. Section 2 defines the problem of model-checking of cyclic programs. Sections 3 - 5 define syntax and semantics of the control software systems. Section 6 defines the cycle abstraction and demonstrates its soundness. Section 7 describes an implementation of the cycle abstraction algorithm in the framework of integrated software design and model-checking. The effectiveness of cycle abstraction is demonstrated in Section 8 that shows the verification results of the NASA robot controller system. Section 9 concludes the paper and positions the cycle abstraction with respect to the existing abstraction techniques.

2 Model Checking of Cyclic Programs

The control flow graphs and 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 cyclic paths in the control flow graph are usually determined by conditional statements (guards) which depend on a subset of the control flow

variables (cycle flow variables). Model checking of such systems generates a traversal of the cycles in the control flow graph for each possible value of each cycle flow variable in the conditional statements which determine the cycles. Each traversal of the cycle with different values of the cycle flow variables is distinct in the state graph of the control system. Additionally each traversal of a cycle will typically have different values for many variables which do not determine paths in the control flow graph. Let us call such variables, "don't care" variables. Each execution of a cycle with different values for "don't care" variables is also distinct in the state graph of the executing system and therefore enlarges the state space for model checking the program.

Many control flow properties are dependent only on the static control flow graph of the system and are independent of the number of traversals of the cycles of the control flow graph. Such control flow properties of a system can be verified by model checking of an abstracted program which has the same static control flow graph as the original (concrete) program.

3 Programming Model for Control Software Systems

Control software systems are often constructed as compositions of sequential programs which interact through sending of messages or events. We can, without loss of generality, assume that each program is comprised of single entry blocks [9].

Definition 1 [Single Entry Block]: *A* single entry (or basic) block *is a sequence of statements which can be entered only at the statement which is at the head of the block and which, when initiated always runs to completion.* For simplicity from now on we refer to a single entry block as an *atom*.

Each program has a FIFO queue for receiving messages (events). Each atom of a sequential program is enabled for execution by arrival of a specific message (event). Execution of an atom may result in sending a message(s) to the program containing the executing atom or some other sequential program. All control flow, both control flow among the atoms internal to a program and among programs, is message (event) driven. Each "send message" statement is conceptually the action of a guarded command over some (control flow) variables.

Definition 2 [Program]: *A sequential* program *is defined as follows:*

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

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

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

compound commands:

$$Proc1,Proc2 \mid if\ B\ then\ Proc1\ else\ proc2\ fi \mid while\ B\ do\ Proc1\ od \mid$$

communication commands:

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

'Generate(ID,exp)' | *'Receive(ID,x)'* .

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

Events generated during the execution of an atom are the *outputs*. In the case if several output events are guarded by the same guard the corresponding outputs are defined as sets of events, one output per branch of the guard.

Definition 3 [**Output**]**:** *An* output *of an atom is an event or a sequence of events for non-guarded or guarded generation of multiple events respectively.*

Each output guard can be composed of nested conditional statements which define different outputs.

The execution model for a *sequential program* is: a) A message arrives in the input queue of a sequential program and some atom of the program is enabled for execution in "run to completion" mode. b) The enabled atom is executed. c) Execution of an atom may result in messages being sent to the process containing the executing atom or to other programs. d) At the end of the execution of a single entry block the program halts and awaits arrival of its next message.

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

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

Definition 5 [**Control Flow Graph of the System**]**:** *The nodes* of the control flow graph *of the system are statements of the sequential programs from which the system is composed. The arcs of the control flow graph of the system connect each statement with its predecessor and successor in the execution of the system.*

Most control properties can be and are stated in terms of control at the single entry block (atom) level.

Definition 6 [**Atom Control Flow Graph**]**:** *The nodes of* the atom control flow graph *of the system are atoms of the composing sequential programs. The arcs of the control flow graph of the system connect atoms 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.

Definition 7 [**Cycle in an Atom Control Flow Graph**]**:** *A* cycle *in an atom control flow graph of a system is repeated execution of a path which begins with the generation of a unique event by an atom and ends at that same atom (cyclic atom).*

We refer to the sequences of events that are repetitively executed, as *redundant* with respect to the verification of control properties.

Each cycle is controlled by a set of output guards that consists of the output guards of the cyclic atom and their dependence set. Let us call the variables of the output guards that define the cycle, *cycle flow variables.*

An algorithm which constructs an abstract system with the same atom control flow graph as the original system is given in Section 6. The algorithm is a source to source transformation of the atoms which preserves all of the outputs of execution of the atoms.

4 xUML - An Instance of the Programming System

xUML [23] is an instantiation of the programming model described above. 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].

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 that we are using is a C-based language extended by the event generation and state machines manipulation commands. It supports standard C types for declaration of the system variables, nondeteministic (*iuniform*) and conditional (*if-then-else/while*) variable assignments, and arrays and enumerated types. Each state machine has a state *labeling variable* which updates imidiately follow the states which receive events. 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 an atom 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 execution model for xUML is *identical* to the execution model for the programming model defined and described in Section 4.

A sample, CONSUMER-PRODUCER-type, xUML program is shown in **Figure 1**. An xUML system is represented by two xUML state machines, a CONSUMER and a PRODUCER. Each state machine is represented by atoms communicating via events. For example, an *Atom D* of the CONSUMER state machine represents an atom that can be activated by an input event *e4* or *e2* and labeled by the update of a variable *status := IDLE.* (In the example, the label variables update commands are implicitely implemented by the xUML graphical development environment.) The activation of the atom is followed by the execution of local commands 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.

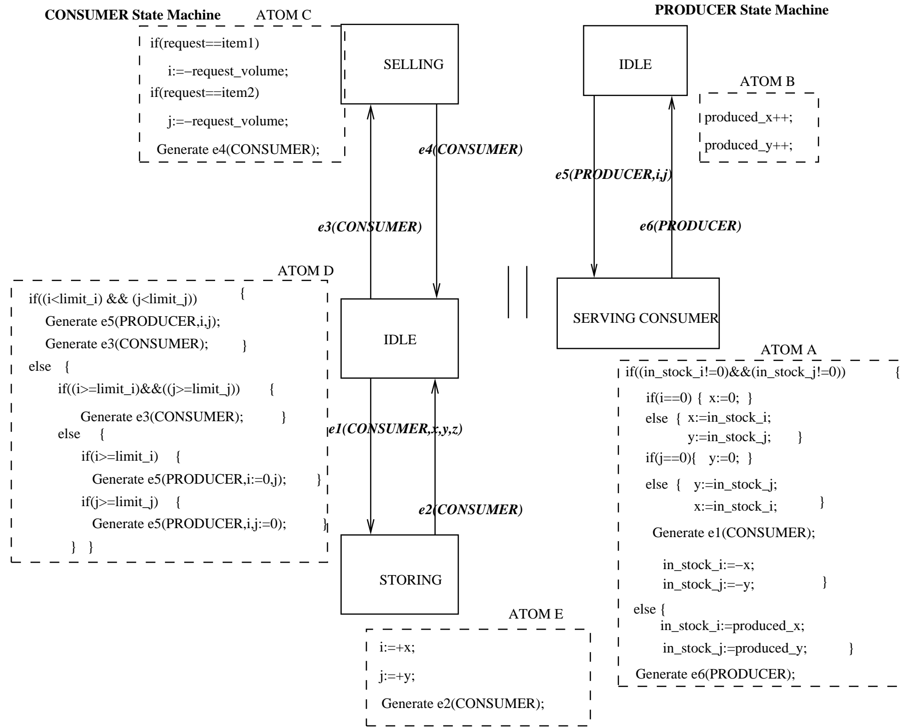


Fig. 1. The CONSUMER-PRODUCER xUML Program

Definition 8 [Control Flow Graph of an xUML System]: *The nodes of* the control flow graph for an xUML system *are the labeled states which receive events. The arcs of the control flow graph of an xUML system connect labeled states which receive events.* The control flow graph for an xUML system can therefore also be specified as a sequence of unique events.

Definition 9 [The xUML Cycle Marker]: Each cycle in the control flow graph of an xUML system is identified by placing a "Cycle Marker" on the initial atom of the cycle.

5 Computational Model

The programming model of Section 3 (and thus also xUML) 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 10 [**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 11 [**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.*

The execution model of a *state machine of an ATS* is: a) An input event arrives in the input queue of a state machine. b) State transitions, including possibly generation of events, are executed until a state requiring input of an event is reached. c) The state machine halts and awaits arrival of its next event.

Definition 12 [**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 13 [**Asynchronous Transition System**]:(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.*

The execution model for the *ATS* is: a) One state machine from among those which are enabled for execution (those state machines with events in their input queues) is randomly selected for execution. d) The selected state machine is executed until it halts awaiting an input event and step *a* is repeated.

Definition 14 [**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 are fair, and only those traces 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 *control trace is fair* if the fairness condition is true in infinitely many states along the trace.

Definition 15 [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.

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

Definition 16 [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.

The actions of xUML state machines execute in run to completion mode. Therefore R.L(C) and R.L(A) correspond to the control flow graphs of xUML systems C and A and L(C) and L(A) correspond to the traces of xUML systems C and A.

6 Cycle Abstraction

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

We present the abstraction *informally* by specifying the cycle abstraction algorithm. We demonstrate the soundness of the abstraction *formally* by presenting a proof of correctness of the cycle abstraction.

6.1 Cycle Abstraction Algorithm[2]

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

Output Range Computation. Figure 2 presents a sketch of the algorithm, which determines the number of outputs guarded by an atom. The algorithm *compute_range* performs syntactic analysis of the guard structure by parsing the text of

[2] We present the cycle abstraction algorithm and its components using the syntax for xUML programs. The cycle abstraction method is, however, a general technique and can be applied to other programming languages. Implementation may then require conducting a trivial static analysis for identification of the output guards that determine the cycle control flow. We assume that an atom that defines a cycle is labeled by the *cycle marker*. The algorithm is implemented in C using string interpretation functions and refers to the xUML-specific syntax structures ('**{ }**', ;) for the conditional statements blocks and commands separation respectively.

the guard and searching for the *Generate* and *if, while* keywords that correspond to the event generation commands and the nested conditional statements, respectively. The conditional statements that guard the event generation commands are counted and the count is stored in the *range* variable. The *compute_range* algorithm 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. The output range computation for an *Atom D* is illustrated in **Figure 3**. The control flow graph (at the command level) illustrates the control flow paths that determine *four* outputs of *Atom D*.

```
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.   While (the end of the body of the conditional statement 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

Guard Transformation. In the abstract program, the output guards of atoms which determine the nodes of the cyclic control flow paths are substituted with multi-way selector expressions, *Choice Selectors*, each of which non-deterministically selects the outputs to be generated during an execution of the atoms. Each *Choice Selector* is defined over a single variable, a *selection variable*, with a range defined by the number of the outputs controlled by the corresponding guard. Each output controlled by the *Choice Selector* selected by a single value of the *selection variable.* Subject to the fairness constraints specified for all values for each *selection* variable, the global state transition graph of the abstract program will have all of the event sequences and thus interleavings of atom executions as the global state transition graph of the concrete program. The atom code is copied command by command with the conditions of the cycle control flow guards replaced by equality comparison of the *selection* variable to one value in its range.

An example of an output guard transformation taken from the PRODUCER-CONSUMER example is given below. The right side represents the original text of *Atom D* and the left side demonstrates the result of the syntactic transformation.

```
selection := iuniform(1,2,3,4);      if((i<limit_i) && (j<limit_j)) {
if(selection == 1) {                       Generate e5(PRODUCER,i,j);
  Generate e5(PRODUCER,i,j);               Generate e3(CONSUMER); }
```

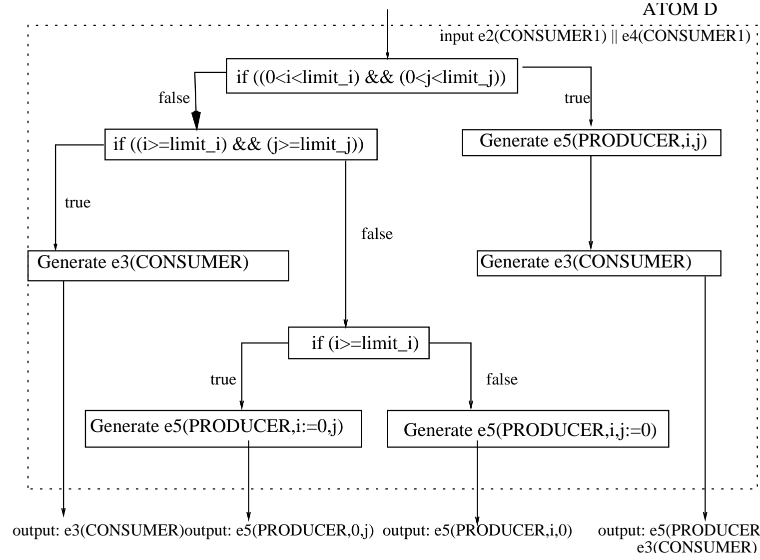


Fig. 3. A Control Flow Graph of Atom D of the Consumer-Producer Program

```
  Generate e3(CONSUMER);  }
else {
  if(selection == 2) {
     Generate e3(CONSUMER);  }
  else {
     if(selection == 3) {
       Generate e5(PRODUCER,i:=0,j);}
     if(selection == 4)  {
       Generate e5(PRODUCER,i,j:=0);}
     } }
```

```
else {
   if((i>=limit_i) && (j>=limit_j))
   Generate e3(CONSUMER);  }
  else {
      if(i>=limit_i)
    Generate e5(PRODUCER,i:=0,j);
      if(j>=limit_j)
     Generate e5(PRODUCER,i,j:=0);
      } }
```

The Cycle Abstraction Algorithm. The cycle abstraction algorithm performs syntactic transformation of the program text by *transformation of atoms*, which execution defines the cycle control flow. The abstraction algorithm starts from the transformation of a cyclic atom by substituting its output guards with the *Choice Selector* expressions. 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 selection variables. To overcome this imprecision, abstraction with respect to cyclic control flow is propagated to the control flow statements which depend on the cyclic control flow variables. A sketch of the algorithm is presented in **Figure 4**.

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

The algorithm proceeds as follows:

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

Step 2: The algorithm iteratively analyzes and transforms atoms that define cycle control flow[3].

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

a) Placement of the first command found in the text of the atom 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 *cycle flow 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*[4] *selection*[j] $:= i$ }[5].

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 atom is not reached then the next command of an atom is analysed. (If the previous command was a guard that was transformed, then the program starts from the command that follows the new structure.)

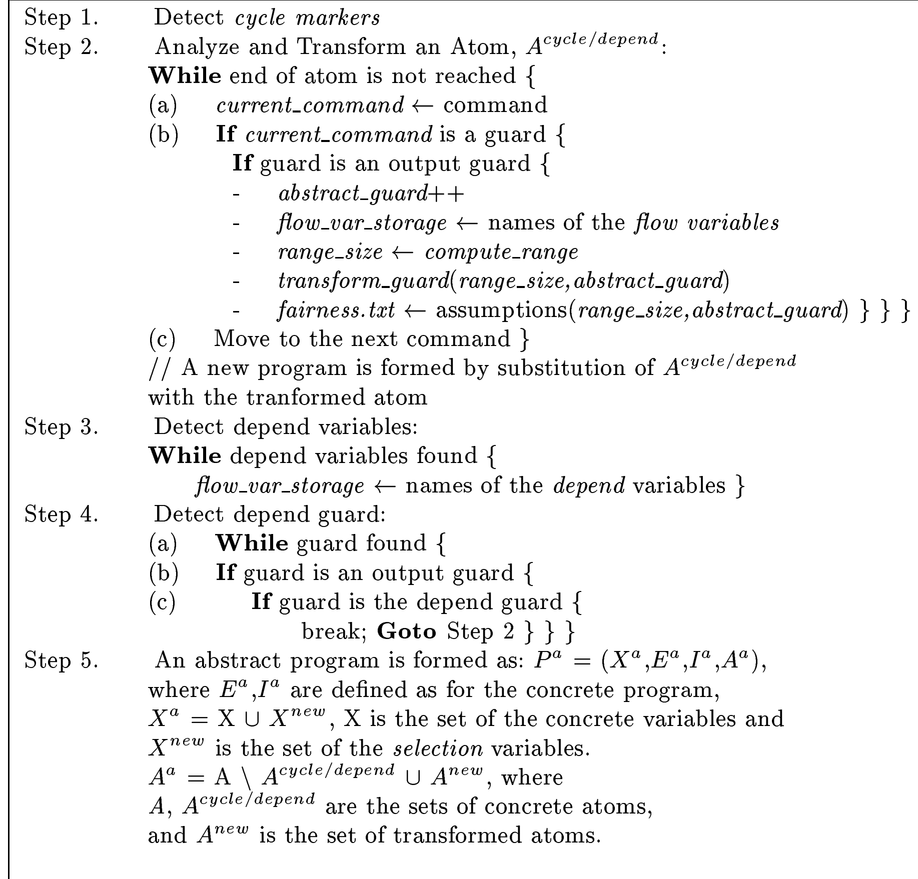
Step 3: Dependency analysis between the program variables and the variables stored in the *flow_var_storage* is performed. The names of the dependent variables are passed to the *flow_var_storage* variable.

---

[3] The atomic structure of the xUML programs is explicitely preserved by special words 'state' and 'endstate' for the beginning and the end of the atom respectively. This allows syntactic identification of the code that corresponds to each xUML atom.

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

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



Step 1. Detect *cycle markers*

Step 2. Analyze and Transform an Atom, $A^{cycle/depend}$:

While end of atom 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 *flow 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 $A^{cycle/depend}$ with the tranformed atom

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: $P^a = (X^a, E^a, I^a, A^a)$,

where E^a, I^a are defined as for the concrete program,

$X^a = X \cup X^{new}$, X is the set of the concrete variables and

X^{new} is the set of the *selection* variables.

$A^a = A \setminus A^{cycle/depend} \cup A^{new}$, where

A, $A^{cycle/depend}$ are the sets of concrete atoms,

and A^{new} is the set of transformed atoms.

Fig. 4. A Sketch of the Cycle Abstraction Algorithm

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.

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

6.2 Correctness of the Cycle Abstraction

The proof of the cycle abstraction uses a *witness* program.

Definition 16 [The Witness Program]: *Let $P^c = (X^c, E^c, I^c, A^c)$ be the concrete program. The witness program, $P^w = (X^w, E^w, I^w, A^w)$, is derived from the concrete program by substituting ALL output guards with non-deterministic* Choice Selector *expressions following the procedure presented in Section 6.1 for the guard transformation.* E^w, I^w are defined as for the concrete program. $X^w = X \bigcup X^{selection}$, where X is a set of variables of the concrete program and $X^{selection}$ is a set of *selection*

variables corresponding to the number of output guards in the concrete program. A^w is a set of the witness program atoms that are the transformed atoms of the concrete program. The transition system corresponding to the witness program is constrained by fairness conditions specified for all values of each *selection* variable. The witness program is a control approximation of a concrete program. The ATS of the witness program has all the interleavings of atom executions as the global transition graph of the concrete program.

Theorem 1:

A transition system of the cyclic concrete program (C) weakly refines control of the transition system of the abstract cyclic (A) program.

Proof Sketch:

The claim is proved by constructing and comparing control traces of transition systems corresponding to *concrete* (C), *abstract* (A) and *witness* (W) programs.

1. We construct a *witness* program from a concrete program following Definition 16.

2. We construct an *abstract* program by applying a cycle abstraction technique to the concrete program.

3. We apply the reduction operator, R, (see definition 15) to generate the control structure of transition systems corresponding to *concrete*, *abstract* and *witness* programs.

4. The control structure of the *witness* program is compared with both the control structures of the *concrete* and *abstract* programs. The following conclusions can be derived from the comparison:

4.1. $X^C \subset X^W$ and R.L(C) $\subset$ R.L(W) by definition of the witness program.

4.2 $X^A \subseteq X^W$ and R.L(A) $\subseteq$ R.L(W) because the cycle abstraction is the *selective* (with respect to cycles) abstraction and the witness abstraction is the *complete* abstraction of the *same* program.

5. As C weakly refines control of W (from 4.1) and A weakly refines control of W (from 4.2), we conclude that C weakly refines control of A.

Therefore, we demonstrated that the cycle abstraction is sound with respect to the control flow representations of the programs. The soundness result implies that a control specification (a property specified in terms of states that receive events) holds for the original program if it holds for the abstract program. Some loss of precision of data computations is introduced by the cycle abstraction. It is traded for the ability to conduct *practical* verification of the control properties of complex control algorithms.

7 Implementation of the Cycle Abstraction

The cycle abstraction has been implemented in the *integrated xUML design and formal verification software development* environment. This section presents the components of the environment and the *cycle abstraction procedure* developed for it.

The Integrated Design and Model-Checking Software Development Environment. This research uses a software development framework that inte-

grates xUML modeling, testing and automata-based model-checking. We refer the reader to [22], [25] for the detailed description of the integrated environment.

The Abstraction Procedure. The steps of the cycle abstraction procedure as they are implemented in the integrated design and verification environment are captured in **Figure 5**. The abstraction procedure operations are supported by the following tools (each tool is represented with respect to the actions it performs in the cycle abstraction procedure):

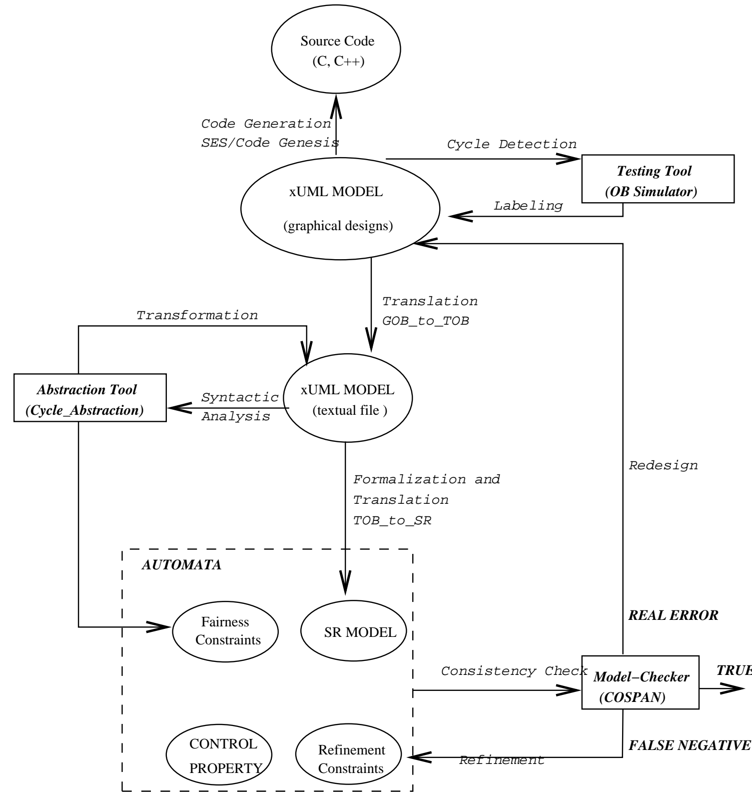


Fig. 5. The Cyclel Abstraction Procedure For the Integrated Design and Model-Checking Software Development Environment

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

- *cycles* 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;

- the *atoms* that are identified to be repeatedly activated are *marked* manually in the xUML specification environment.

2. *The* CYCLE_ABSTRACTION *program:*[6]

- the labeled xUML state machines are *syntactically analyzed* and *transformed* into the abstract xUML state machines using the cycle abstraction algorithm defined in the previous section.

- 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[13]:

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

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

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

A detailed description of the xUML notation and the automata-based model-checking, as it is supported by COSPAN, can be found in [13, 22].

8 Evaluation of the Cycle Abstraction Technique

Test-Bed Software. This section presents the NASA robot controller system (RCS) formulated as xUML models. The RCS is a complex legacy software system used for instantiation of numerous robotics applications that size is roughly 300K LOC. In fact, a part of the RCS has been recently included in the control software system of the robot arm, deployed in the space station [12]. The RCS has been thoroughly analyzed by testing. A number of efforts have been made to conduct formal verification by model-checking. Some success was reported in the compositional verification of a simplified RCS components [22]. The complexity (partially caused by the cyclic execution of the robotic algorithms) of the RCS components, however, prevented practical model-checking of realistic systems. Due to space limitations and the complexity of the test case software, we refer the reader to our previous papers where the application software and its properties are presented in detail [22, 21]. In this paper we refer to a subset of the RCS, namely the *Kinematics* unit, which 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. For a set of target positions of

[6] The abstraction procedure uses a translator that automatically transforms the xUML programs from the Graphical OB representation into SR, an input language of the model-checker, COSPAN [25]. Specifically, the CYCLE_ABSTRACTION program is applied to the intermediate representation of the translation result, the textual representation of the xUML models.

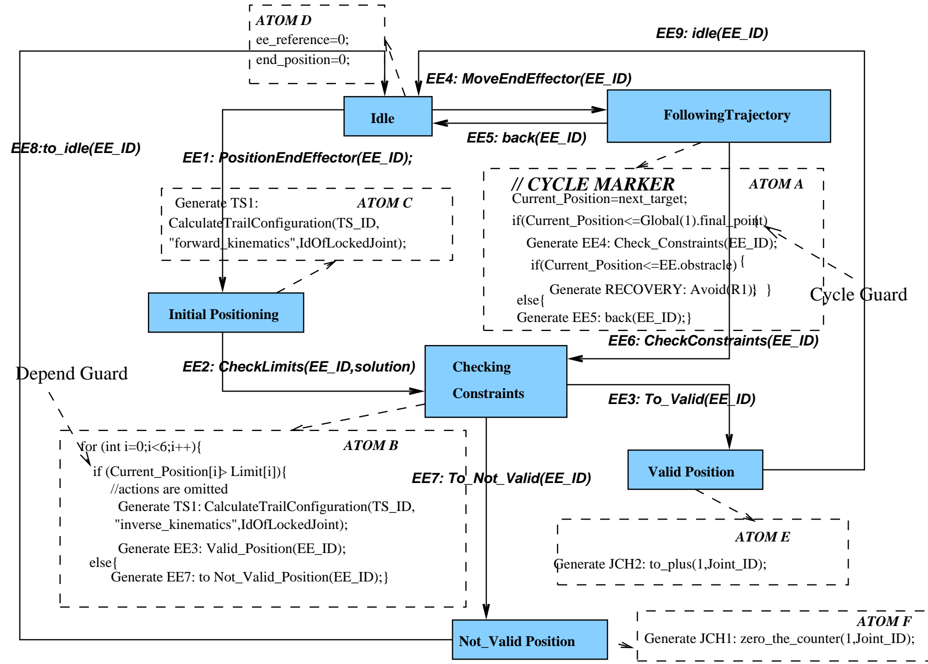


Fig. 6. xUML state machine of the End_Effector program of the Robot System.

the robot arm, the same set of actions leading to the calculation of the joint angles and to the check of the physical constraints satisfaction is required.

- *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.

The *Kinematics* module of the RCS is modeled by the state machines, representing behavioral specifications of the *Arm*, *Joint*, *End_Effector*, *Checker*, *Recovery*, *Trial_Configuration*, *Global_ Representation* xUML objects. **Figure 6** shows the state machine of the *End_Effector* process as an example of an xUML behavioral specification.

Experimental Results. We considered several variants of the test-bed system with a different number of joints i instantiated for a single robot arm. We checked a number of the *control properties* for various configurations of the robot arm. Due to the space limitations in this paper we present verification results of one property that representively demonstrates the complexity reductions achieved by application of the cycle abstraction. In English the property is state as follows: is it possible

for the end-effector to proceed to a new target position when an obstacle has been reached by the robot arm or a joint(s) have encounted a faulty configuration? 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 in the xUML system.[7] The formal specification of the above property is (the property is encoded in the query language of COSPAN):

$$Always(End_Effector.status = \text{'FollowingTrajectory'} \rightarrow Arm.status = \text{'Valid'}).$$ [8]

We used two models to check the above property. 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 cycle abstraction method has been applied. The robot controller system abstraction was instantiated by the detection of two cyclic atoms of the *Joint* and *End_Effector* state machines. One of the cyclic atoms is the *Atom A* of the *End_Effector* state machine as shown in **Figure 6**. The cycle abstraction was enforced by substitution of the output guards of the cyclic atoms by *Choice Selector* expressions. For example, an output guard of the *Atom A*,

Table 1. Comparison of Verification of the Concrete and Abstract Robot Controller Systems

Value of i	Concrete Model (states/secs/Mbytes)	Abstract Model (states/secs/Mbytes)
2	11,933/1,452/1.81	97/25.39/0.3
3	26,119/7,966/5.03	229/69.9/0.31
4	102,067/56,414/18.7	1,105/817.8/1.2
5	memory exhaustion	10,389/1,211/2.5
6	memory exhaustion	32,518/5,132/7.25

(*if(Current_Position != Global(1).final_point)*), that determines three outputs of the atom has been substituted with the *Choice Selector* expression defined over a *selection* variable that ranges between 1 and 3. The correctness of the control paths of the abstract robot controller system with respect to the modified output guard, was preserved by transformation of the output guards that depend on the variable *Current_Position* directly and indirectly through a chain of dependencies. For example, an output guard of *Atom B* has been transformed as well.

The dependency analysis and the abstraction mapping along with the generation of fairness constraints for each cyclic output guard was conducted *automatically* by the CYCLE_ABSTRACTION tool. For example, the following set of fairness constraints was generated for the *selection variable* of the *Choice Selector* used in *Atom*

[7] The labeling variables values are preserved by the cycle abstraction since they do not depend on any program variables but the fact that an event arrives to an atom.

[8] The char type of the *status* variables is intepreted into the integer type by the OB-SR translator to confirm to the model-checker.

A: *AssumeEventually (selection[0] = 1); AssumeEventually (selection[0] = 2); AssumeEventually (selection[0] = 3).*

When false negative results were encounted as a result of model-checking, additional behavioral restrictions specified in terms of the selection variables were added to the list of the assumptions. The assumptions were derived from the domain knowledge of the RCS acquired during the simulated testing supported by simulation tool [19].

Table 1 compares the run-time and memory usage for the concrete and the abstract RCS with a total number of 7 processes excluding the i processes 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 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. The verification for the robot configurations consisting more than 4 joints could not be completed due to the memory exhaustion for the concrete model, COSPAN succeeded for the abstracted model.

9 Summary and Related Work

Summary. This paper gives a data abstraction technique application of which results in efficient model-checking of cyclic control-intensive software systems. We demonstrated that the cycle abstraction algorithm produces a conservative abstraction with respect to the control traces of the concrete program. The cycle abstraction method applies for asynchronous interleaved sequential programs that are the common modeling representation for control software systems. The cycle abstraction algorithm has been implemented for xUML software systems. The xUML notation supports separation of data and control. This separation enables syntactic identification of the decision points that determine control for the cyclically executing programs. The cycle abstraction method is, however, a general technique and can be applied to other programming languages. Implementation may then require conducting a trivial static analysis for identification of the output guards that determine the cycle control flow.

The cycle abstraction method has been implemented in the integrated design, testing and model-checking environment supported by the commercial tools, SES/Objectbench [19] and automata-based model-checker, COSPAN [8] used in previous studies [21, 22, 25].

We evaluated the cycle abstraction technique by verification of a NASA robot controller system. Order of magnitude reductions in both time and space were found for model checking the abstract program compared to the concrete program.

Related Work. Data abstraction has long been a favored method for reducing the state-space of a software system to allow efficient model-checking [4, 6, 16]. Abstraction techniques are often based on abstract interpretation [5] and require a user to give an abstraction function relating concrete datatypes to abstract datatypes. Predicate abstraction was been introduced by S. Graf and H. Saidi [7] and has been

widely accepted as the basis for automated abstraction tools [1, 15, 18]. Cycle abstraction is similar to predicate abstraction in that it requires specification of an abstraction function as predicates over concrete data. Cycle 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 cycle 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 cycles. Cycle 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 the predicate abstraction methods. Cycle abstraction is a useful complement to the predicate abstraction techniques. It abstracts control while predicate abstraction abstracts statements not effected by the cycle abstraction. We are currently engaged in a project that develops a prototype automatic abstraction tool supporting the predicate abstraction algorithm presented in [15]. We are planning to evaluate the cycle abstraction in combination with the predicate abstraction.

The implementation of the cycle abstraction algorithm is similar to [15] in that the cycle abstraction algorithm does not construct the explicit state graph either of the original or of the abstract programs. Instead a syntactic analysis of the original program is used to produce an abstract program. However, our approach is different from any other abstraction algorithms dealing with the source code in that the abstraction is applied to a design-level specification. 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). Cycle abstraction provides complete coverage of all possible cyclic execution paths. 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 is that many potential paths are impossible to reach because of data relationship constraints. Cycle 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)