Translating Ada Programs for Model Checking : A Tutorial

Matthew B. Dwyer, Corina S. Pasareanu
Dept. of Computing and Info. Sciences
Kansas State University
Manhattan, KS 66506
{dwyer,pcorina}@cis.ksu.edu

James C. Corbett
Dept. of Info. and Computer Science
University of Hawai'i
Honolulu, HI 96822
corbett@hawaii.edu

October 2, 1998

1 Introduction

This document is a tutorial introduction to a toolset for translating Ada source code to the input format of the SPIN model checker [Hol97] (i.e., Promela code) and the SMV model checker [McM93].

This toolset provides completely automated translation for most steps in the process of generating a safely approximating state transition model of a software system's run-time behavior. This process, as supported by the toolset, is illustrated in Figure 1. There are seven steps in model checking properties of an Ada program.

1. Conversion of the original Ada program to a safe finite-state variant
2. Translation to SEDL
3. Definition of program features to be reasoned about
4. Translation to model checker input
5. Property specification using defined features
6. Execution of the model checker
7. Interpretation of counter-examples for false results

The toolset provides support for steps 2-7, which we discuss in detail in this paper. Step 1 is not considered here—the interested reader is referred to [Cor98, DHN98, DP98a, DP98b, HDL98a, HDL+98b] for techniques which can convert general Ada programs to a finite-state variant.

The Ada programs that can be fed to step 2 can be thought of as *static finite-state* Ada programs. The Quechua [Cor93] language defines the data types and operations that such an Ada program may contain (although Quechua's syntax differs slightly from legal Ada). Finite-state Ada programs have a fixed number of tasks, they contain no pointer-based referencing of tasks, and their data is defined over a discrete finite type (e.g., sub-ranges, enumerations, booleans). We refer the reader to Quechua for details of legal input Ada programs.

It is well-understood that cost-effective model checking of complex systems requires abstraction [CGL94, Dam96, LGS+95]. One crucial issue in translating source code to a highly-abstracted finite-state transition system for model checking is insuring the correspondence between the features of the

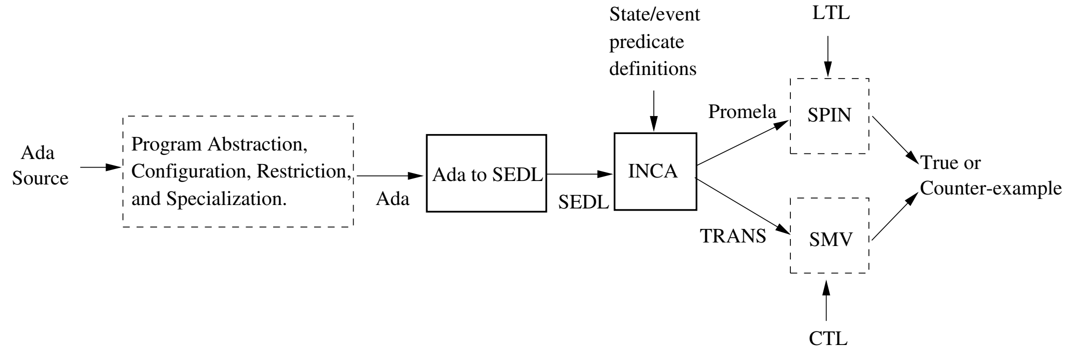

Figure 1: Ada Translation Toolset

source code that are to be reasoned about and features of the transition system to be checked. For example, one may be interested in reasoning about sequences of calls to procedures in a program (e.g., that calls to `open` are always followed by calls to `close`). Translations must reveal the states of the transition system that correspond to the program features of interest (e.g., `open` calls). Furthermore, a mechanism that allows these features to be referred to in property specifications that are to be checked must be provided. The toolset described here defines just such a mechanism; it is referred to as *predicate definition*, even though it provides a means for defining the set of proposition to be used in temporal logic specifications. This predicate definition facility is used in step 3 by the tool user and in step 4 by the translation toolset. One nice feature of the predicate definition facility is that it is independent of the particular model checker to be used. The same predicate definitions can be used to define the propositions for checking a CTL [CES86] specification with SMV or an LTL [MP91] specification with SPIN in step 5.

Once the model checker has completed execution, in step 6, its results need to be interpreted, step 7. A false result will be accompanied by a counter-example, i.e., a path through the transition system that violates the specified property. Model checkers express counter-examples in terms of sequences of transition system states. This is problematic for programmers who are familiar with the code, but not the details of the translations produced by the toolset. By incorporating predicate definitions in step 4, the tools allow counter-examples to be expressed in terms of source program features that the user is interested in, thereby greatly simplifying the interpretation of counter-examples.

This tutorial is example driven; the complete details of this example, including source code, specifications and intermediate files, have been collected on the web-site:

`http://www.cis.ksu.edu/~dwyer/ada-modelcheck`

We begin with an overview of how a simple property of a program is checked. In the next section, we define the underlying model of concurrent programs that the translation tools are based on. We then proceed with a detailed discussion of steps 2-7. We conclude with a description of features that might be included in future generations of similar translation toolsets and a summary.

2 Overview

Consider the classic readers writers problem (as described in e.g., [ABC+91, Cor96, Cha96]) and a simple invariance property stating that: *writers cannot work while some reader is active.* We

```
(defpredicate "callwritework"
              (after "internal(writer_1;callwritework)"))
(defpredicate "activereaders0"
              (in-task read_write_control (= activereaders 0)))
```

Figure 2: Predicates for Readers-Writers Invariance Property

might specify this in LTL as `[](callwriteWork -> activereaders0)`, where `callwriteWork` indicates that a writer's procedure `writeWork` is called and `activereaders0` indicates that the variable `ActiveReaders` in the `Read_Write_Control` task has a value of zero.

2.1 Step 2

The finite-state Ada code for the 2 reader-1 writer program is given in a file `rw21.a`. Executing the command

```
$ ada_to_sedl rw21
```

produces the file `rw21.sedl`.

2.2 Step 3

We define predicates for the propositions that appear in our specification by putting the `defpredicate` forms in Figure 2 in a file named `rw21.query`.

2.3 Step 4

Invoking the rest of the translation tools requires starting the LISP environment. Once LISP is running the INCA toolset is loaded, then the `derive` and `spin-input` commands are executed.

```
$ lispworks
...
USER 4 > (load "inca.lisp")
USER 5 > (derive "rw21" :expose '("writeWork"))
USER 6 > (spin-input :embed '("callwriteWork") :define '("activereaders0"))
USER 7 > (bye)
```

These commands produce a SPIN input file `rw21.prom`.

2.4 Step 5

The LTL specification `[](callwriteWork -> activereaders0)` is submitted to SPIN to render it in a form (i.e., a never claim) that can be included into `rw21.prom`.

2.5 Step 6

Executing SPIN requires several commands.

```
$ spin -a rw21.prom
$ gcc -o pan pan.c
$ pan -a
```

The results of executing `pan` indicate a successful model check (i.e., the property is true of the program) so there is no counter-example to interpret and hence no step 7.

3 Example Ada Program and Properties

Versions of the readers writers problem are readily available as static finite-state Ada programs. We have selected a version with two reader tasks and one writer tasks. We have added procedure `readerWork` and `writerWork` which are called from the reader and writer tasks, respectively. The source code for this system is given on the web-site.

We check the following collection of correctness properties for this system. For each property, we describe it in English, as a specification pattern [DAC98b, DAC98a], as LTL and as CTL.

1. *A writer task performs its work when there are no active readers.*
   This is a global-universal property which is
   `[](callwriteWork -> activereaders0)` in LTL and
   `AG(callwriteWork -> activereaders0)` in CTL.

2. *A writer starting will eventually be followed by a writer stopping.*
   This is a global-response property which is
   `[](callstartwrite -> <> callstopwrite)` in LTL and
   `AG(callstartwrite -> AF callstopwrite)` in CTL.

3. *A writer starting will eventually be followed by a reader starting.*
   This is a global-response property which is
   `[](callstartwrite -> <> (callstartread1 || callstartread2))` in LTL and
   `AG(callstartwrite -> AF(callstartread1 | callstartread2))` in CTL.

The first two properties hold on the the `rw21` system, but the third one does not. We use this final property to illustrate the interpretation of model checker produced counter-examples.

These specifications are written using a set of six propositions describing different aspects of executable system behavior in terms of features of the source code:

`callwriteWork` Procedure `writeWork` is called.

`activereaders0` Variable `ActiveReaders`, which is local to the `Read_Write_Control` task, has a value of zero.

`callstartwrite` The `Writer_1` and `Read_Write_Control` tasks rendezvous on the `Start_Write` entry.

`callstopwrite` The `Writer_1` and `Read_Write_Control` tasks rendezvous on the `Stop_Write` entry.

`callstartread1` The `Reader_1` and `Read_Write_Control` tasks rendezvous on the `Start_Read` entry.

`callstartread2` The `Reader_2` and `Read_Write_Control` tasks rendezvous on the `Start_Read` entry.

4 Some Terminology and Background

Predicates and their encoding into model checker input are defined in terms of INCA's internal model of concurrent programs as a set of communicating finite-state automata (CFSA)[1]. We define this model to clarify the subsequent definition of predicates. A CFSA is denoted $M_i = (S_i, \Sigma_i, \Delta_i, s_{0,i}, F_i)$ where:

- S_i is a set of *local states*
- Σ_i is a set of *actions*
- $\Delta_i \subseteq S_i \times \Sigma_i \times S_i$ is a *transition relation*
- $s_{0,i} \in S_i$ is the *start state*
- $F_i \subseteq S_i$ are the *final states*

Each task in an Ada program is represented by an automaton M_i. The local states typically represent points of control within a task and may also encode the values of finite-ranged variables that are critical for modeling the program behavior related to the property being checked. The concurrent system modeled by a set of n automata, $M_1, \ldots, M_n$, can be defined as another automaton $M = (S, \Sigma, \Delta, s_0, F)$ where:

- $S = S_1 \times \ldots \times S_n$ are *global states*
- $\Sigma = \bigcup_i \Sigma_i$
- $((s_1, \ldots, s_n), a, (s'_1, \ldots, s'_n)) \in \Delta$ iff $\forall i = 1, \ldots, n$:

$$(a \notin \Sigma_i \wedge s_i = s'_i) \quad \bigvee \quad (a \in \Sigma_i \wedge (s_i, a, s'_i) \in \Delta_i)$$

- $s_0 = (s_{0,1}, \ldots, s_{0,n})$ is the *global start state*
- $F = F_1 \times \ldots \times F_n$ are the global final states

Since we are modeling Ada rendezvous, we may assume that tasks synchronize in pairs and thus each symbol $a \in \Sigma$ is in the alphabet of either one or two of the M_i. Symbols in the alphabet of a single M_i are called *internal actions* and represent the execution of code within the corresponding task; symbols in the alphabets of two M_i are called *communication actions* and represent a rendezvous between the corresponding tasks. Ada allows accept statements to have bodies where the accepting tasks can execute while the calling task is blocked. This is modeled by a pair of rendezvous one for the start and one for the end of the accept body.

[1] We use CFSA for both the singular communicating finite-state automaton and the plural communicating finite-state automata.

5 Translating Ada To SEDL (step 2)

Ada is converted to SEDL using the IRIS-Ada toolset [FSZ88]. This consists of typical compiler front-end processing, the generation of control-flow graphs for tasks and procedures in the program, and finally emitting of SEDL. The shell script `ada_to_sedl` acts as a driver for this sequence of tools [2]. The script accepts a list of basenames for Ada files with the `.a` suffix. For the readers-writers example the command:

```
$ ada_to_sedl rw21
```

produces the file `rw21.sedl` (the complete file is available on the web-site).

Known bugs in the implementation of `cfgs_to_sedl` include:

- task header syntax errors
- procedure syntax errors
- improper modeling of accept bodies

Currently, hand editing of the sedl file is sufficient to workaround these problems. The SEDL on the web-site illustrates the edited readers-writers example's SEDL with indications of the changes that were made.

6 Defining Predicates (step 3)

In this presentation, we interleave the description of the syntax and semantics of predicate definitions. Predicates define collections of local states in a CFSA model of a program that share some common feature. Features can include, calls to or returns from specific procedures in the program, the occurrence of a rendezvous communication between tasks in a program, the occurrence of a named internal action in the CFSAs, or program variables attaining specified values.

All predicates have a name. These are the proposition names that will be used in LTL/CTL specifications submitted to the model checker. Naming conventions in model checker input languages require that the predicate names begin with a lower-case alphabetic character.

<predicate definition>	→	"(" "**defpredicate**" <id> <predicate> ")"

Sometimes we are interested in reasoning about the sequencing of program events rather than the sequencing of program states. The distinction may seem subtle at first, but it can have a significant impact on the specification one writes. While it is true that events trigger state transitions, if one is interested in reasoning about a state (e.g. `x>3`) it can be very awkward to specify that state using events (e.g., one would have to interpret the events that modified variable `x`). On the other hand, if one is interested in reasoning about a sequence of events (e.g., the number of times an elevator arrives at a floor) it can be awkward to specify this in terms of transitions in state values (e.g., counting transitions from `atfloor` being true to being false and vice versa). For these reasons, both *event* and *state* predicates are supported.

<predicate>	→	<event predicate>
	|	<state predicate>

[2] This script was previously called `run-cfgs_to_sedl`

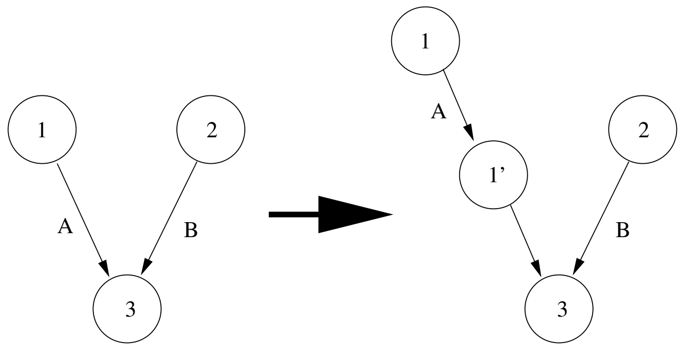

Figure 3: Internal Events and FSAs

6.1 Event Predicates

A limited set of program events can be represented in predicate definitions: user-defined events that have been added to the system and rendezvous communication. Note that procedure call and return events are currently supported through a different mechanism (see `:expose` below).

An event predicate is true if and only if any task containing the event specified is in a state immediately following a transition on that event. For example, state 3 in the CFSA on the left side of Figure 3 immediately follows a transition on `A` or `B`. Note that this does not necessarily mean that the event just occurred—other tasks may have made transitions since the event (i.e., another CFSA made a transition). In fact, if there are multiple transitions into the state, the event may not have occurred at all since the state may have been entered on a transition not labeled with the event (e.g., even though state 3 is after a transition on `A` it could be entered by a transition on `B`). If you wish to uniquely define the occurrence of an event you should embed rather than define the predicate (see `:embed` below).

<event predicate>	→	“(” “**after**” <event string> “)”
<event string>	→	“"” <id> “"”
	|	“"” “**call(**” <id> “;” <id> “.” <id> <params> “)” “"”
<params>	→	“;” <constant> <params>
	|	“;” <constant>

Events are defined by a string (note that white-space in the strings is not allowed even though the grammar allows for them). Strings which consist of only an identifier specify a user-defined events. Such an event, and its associated post-state, are inserted into the CFSA using the `internal` directive in SEDL. The right-side of Figure 3 illustrates the CFSA structure introduced for an `(internal "A")` directive in a branch of a conditional that has a `B` action in its other branch. An event predicate will match any occurrence of a user-defined event with the indicated name (e.g., `(after "A")` is true if any task is in a state following an `(internal "A")` action). Note that the CFSA generated for a system containing an `internal` action will contain only one transition (e.g., labeled with `"A"`) into the states where the corresponding `after` event predicate (e.g., `(after "A")`) is true. In this case, the `after` predicate does guarantee the occurrence of its event since no other event leads to that state (e.g., in the figure event `B` does not lead to state $1'$)

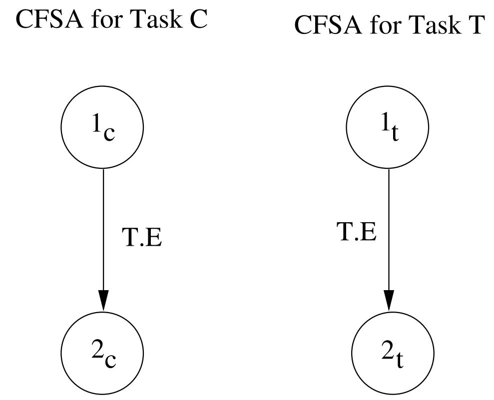

Figure 4: Communication Events and FSAs

Strings which begin with the `call` prefix correspond to pre-defined events which are generated to model rendezvous. Such an event corresponds to simultaneous transitions in two CFSA on a shared symbol. Figure 4 illustrates such a situation. A predicate definition that matches such an event is encoded as a string using the names of the participating tasks and the name of the entry (e.g., `(after "call(C;T.E)")` is true in states in which task C has just completed a rendezvous with task T at entry E). Entry calls can have parameters that can be encoded into the call string as a semi-colon separated list; only literal values can be supplied and values for all parameters must be given in the order of the parameters of the call. Output parameters of entry calls with accept bodies are modeled as parameters to explicitly defined call-back rendezvous (details of this await implementation of this capability in `ada_to_sedl`).

The event predicates for the `rw21` example are given at the top of Figure 5.

6.2 State Predicates

A state predicate specifies a task and a boolean expression over the values of that task's variables. As discussed below, one can model each task as a CFSA which interacts with those of the other system tasks. The state predicate is true in states of that task's CFSA where the expression would evaluate to true.

```
<state predicate>  →  “(” “in-task” <id> <expr> “)”

           <expr>  →  <constant>
                   |  <id>
                   |  “(” <op> <expr> ... <expr> “)”

       <constant>  →  “true” | “false” | <digits> | <id>

             <op>  →  “and” | “or” | “not”
                   |  “+” | “-” | “*”
                   |  “index” | “field”
                   |  “>” | “<” | “>=” | “<=” | “=”
```

```
; Event Predicates
(defpredicate "callwritework"
              (after "internal(writer_1;callwritework)"))
(defpredicate "callstopwrite"
              (after "call(writer_1;read_write_control.stop_write)"))
(defpredicate "callstartwrite"
              (after "call(writer_1;read_write_control.start_write)"))
(defpredicate "callstartread2"
              (after "call(reader_2;read_write_control.start_read)"))
(defpredicate "callstartread1"
              (after "call(reader_1;read_write_control.start_read)"))

; State Predicates
(defpredicate "activereaders0"
              (in-task read_write_control (= activereaders 0)))
(defpredicate "writerpresentfalse"
              (in-task read_write_control (= writerpresent nil)))
```

Figure 5: Predicates for Readers-Writers Properties

The expression must be legal Quechua [Cor93]. The tools perform only minimal type checking, so care should be taken. An expression can be tested for legality by placing it in an `(assume <expr>)` statement in the SEDL task specification; the INCA deriver (discussed below) will then perform all necessary checks to insure legality.

Only variables local to a single task may be used in an expression. This localization of predicates is not a significant limitation for specifying properties of systems. For example, if one wishes to describe the system state where variable `x` in task `A` is positive and variable `y` in task `B` is equal to zero, one defines two predicates:

```
(defpredicate "xpos" (in-task A (> x 0)))
(defpredicate "yzero" (in-task B (= y 0)))
```

then uses the conjunction `xpos && yzero` in the temporal logic specification. Because predicate definitions are local, due to the nature of the CFSA model, relational predicates (e.g. `x=y`) cannot be expressed for variables in different tasks.

To check parameter values, use event predicates—the parameter values are encoded in the string for the rendezvous as described above. There is currently no support for checking parameter values of procedure calls or global variables.

State predicates for the `rw21` example include at the bottom of Figure 5.

7 Translating to Model Checker Input (step 4)

The INCA toolset consists of a component called the *deriver*. This component converts a program expressed in SEDL into a collection of CFSA, one for each tasks in the program. Since SEDL programs have variables and CFSA do not, it is the job of the deriver to encode the state of the program's variables and their influence on program control flow into the structure of the CFSA. The deriver achieves this by applying a wide-range of program analysis and transformation techniques[3] including: procedure inlining, loop unrolling, constant propagation, constant folding, and dead-code elimination. These techniques are applied to each program task in isolation and can yield very large CFSA for tasks that manipulate large amounts of data.

The INCA tools are written in LISP; the current implementation uses Harlequin's LispWorks[4]. For the `rw21` system the following commands yield the expanded CFSA for the four system tasks:

```
$ lispworks
LispWorks: The Common Lisp Programming Environment
...

CL-USER 4 > (load "inca.lisp")
; Loading text file inca.lisp
; Loading text file toolset.lisp
; Loading fasl file fe.afasl
; Loading fasl file ig.afasl
; Loading fasl file ilp.afasl
; Loading fasl file spin.afasl
; Loading fasl file smv.afasl
; Loading fasl file cplex.afasl
INCA 3.3 loaded. Type (help) for information.

USER 5 > (derive "rw21" :expose '("writeWork"))
Translation complete
Executing writer_1..reader_2.reader_1.read_write_control.....
Queries:
("derive" :OK 1.1)
```

The quoted basename, `rw21`, of the SEDL input file is given as the first parameter to the `derive` command. The command accepts several optional parameters including the `:expose` and `:compress` options.

The `:expose` option accepts a list (i.e., the leading ' is important) of quoted names of procedures in the program. With this option the deriver models calls (returns) to those procedures using `internal` statements with symbols built from `call_` (`return_`) and the procedure name. The names of the internal statements for our example are `call_writework` and `return_writework`. Reasoning about procedure calls and returns would not be possible without this feature since all procedure calls are inlined by the deriver. It is important to note that use of the `:expose` option does not automatically define event predicates for the new internal statements; those must be defined explicitly through the predicate definition mechanism.

The `:compress nil` option will cause the deriver to skip live variable analysis and a subsequent optimization that automatically sets dead variables to their initial values. This can sometimes be useful when using state predicates.

INCA can output the CFSA for each task built by the deriver, as well as other information related to the derivation process, with the `booklet` command.

[3] One can view the deriver as an aggressive (i.e., maximally polyvariant) partial evaluator for programs using only types with finite domains.

[4] LispWorks is a registered trademark of Harlequin Group plc.

```
USER 6 > (booklet :fsas t)
Booklet written to file rw21.book
:OK
```

Finally, the deriver reads the file with the given basename and a `.query` extension, if it exists. This is the file in which one should place predicate definitions to be incorporated into the CFSA.

A useful feature that is unrelated to the INCA tools is LISP's facility for timing any command. Passing the command as an argument to the `time` command will execute the argument and report timing information.

7.1 From CFSAs to Model Checker Input

INCA provides commands to output Promela programs for use with the SPIN model checker and transition systems for use with the SMV model checker from the collection of derived CFSA; these commands are `spin-input` and `smv-input`, respectively. Model checker input languages differ in their syntax as well as the programming models they support. The approach of the INCA translators is slightly different for the two model checkers. For SPIN, each task CFSA is translated to a Promela process where shared labels in the CFSA are implemented with synchronous communication in Promela. For SMV, the transition relations of the individual CFSA are composed into the global transition relation which is expressed as a TRANS statement.

As mentioned earlier, an important feature of these tools is their ability to encode predicate definitions in the model checker input. The predicate definitions in the `.query` file serve as the default set of definitions. Users can override this default. For example, the predicates for a `spin-input` command can be set with a quoted list of (`<id> <predicate>`) pairs following the `:predicates` keyword:

```
(spin-input :predicates '(("name1" <predicate1>) ... ))
```

7.2 Defining vs. Embedding Predicates

There are two mechanisms for encoding predicate definitions into model checker input. The mechanisms are referred to as *defining* and *embedding* predicates; the user control the mechanism that is used through use of the `:define` and `:embed` parameters to the input translator commands.

The distinction between these approaches can be characterized as follows. Defining a predicate amounts to identifying the set of global states at which the predicate evaluates to true; thus, a defined predicate is a disjunction of equality comparisons of the current state and the states in that set. Embedding a predicate actually modifies the CFSA by adding new system states that are used to model points at which the predicate will become true. This is done by introducing boolean state variables for each embedded predicate and transitions which set and unset it at appropriate places.

Embedding predicates will increase the size of the state space to be model checked, but it is not clear how it will affect the overall cost of analysis when compared to defined predicates. One should not embed unnecessary predicates for the property being verified—the fewer predicates embedded, the smaller the state space. Extra, unused, defined predicates appear to have no affect on performance. This means that users can have a single query file with all of the predicates for a system then selectively embed only those that are needed.

It is important to note that the semantics of the `after` event predicate is different if the predicate is embedded. An embedded event predicate (e.g., `(after "A")`) is true in global states where the event (e.g., `"A"`) just occurred. This mean that the predicate is true for a single step of the transition system, unless the action is repeated. This eliminates the need for additional `internal` statements in the system that are there to insure that the state after an event is unique. Thus, one common strategy for predicate definition is to `:embed` event predicates and `:define` state predicates.

The input translator commands allow one to freely mix embedding and definition. For example, the `smv-input` command:

```
(smv-input :embed '("pred1" "pred2") :define '("pred3"))
```

embeds predicates `pred1` and `pred2` while still defining `pred3`; these predicates would have been defined in the `.query` file. The keyword `:all` is used to `define` or `embed` all predicates listed in the `.query` file. For example:

```
(smv-input :embed :all)
```

The `spin-input` command takes the same parameters.

For the `rw21` example we used the following commands:

```
USER 7 > (spin-input :embed '("callwriteWork" "callstopwrite"
                              "callstartwrite" "call_writeWork")
                     :define '("writerpresentfalse" "activereaders0"))
SPIN input written to file rw21.prom
NIL

USER 8 > (smv-input :embed '("callwriteWork" "callstopwrite"
                             "callstartwrite" "call_writeWork")
                    :define '("writerpresentfalse" "activereaders0"))
SMV input written to file rw21.trans
NIL
```

8 Writing Property Specifications (step 5)

Writing temporal logic specifications for system requirements can be a challenge for a number of reasons, not the least of which is lack of familiarity with the formalisms (e.g., LTL and CTL) used by model checkers. We advocate an approach to writing specifications that involves the reuse of parameterized specification patterns. The approach is described in [DAC98b, DAC98a] and an extensive web-site has been developed to support its application:

`http://www.cis.ksu.edu/~dwyer/spec-patterns.html`

The specifications for the `rw21` example were written using the patterns. Note that the strings used to define predicates can be used as proposition names in LTL and CTL formula without modification.

9 Running the Model Checker (step 6)

SMV and SPIN are flexible tools with a large number of options and configuration parameters. We refer the reader to the the documentation for those tools which can be found at:

`http://netlib.bell-labs.com/netlib/spin/whatispin.html`

for SPIN and

`http://www.cs.cmu.edu/~modelcheck`

for SMV.

```
bit callstartwrite = 0;
bit callstopwrite = 0;
bit callwriteWork = 0;

/* activereaders0 = (IN-TASK READ_WRITE_CONTROL (= ACTIVEREADERS 0)) */
#define activereaders0 (read__write__control[4]@state_11 \
    || read__write__control[4]@state_10 \
    || read__write__control[4]@endstate_9 \
    || read__write__control[4]@state_8 \
    || read__write__control[4]@state_6 \
    || read__write__control[4]@state_3 \
    || read__write__control[4]@state_2 \
    || read__write__control[4]@state_1)
```

Figure 6: Embedded and Defined Predicates for SPIN

9.0.1 Model Checker Input Files

Here we describe only the small portion of the model checker input files that is related to predicate definitions. The complete files are given for SPIN and SMV on the web-site.

SPIN's input language, Promela, is C-like. Promela files can include the C pre-processor directives `#include`, to inline definitions in other files, and `#define`, to establish macro definitions. SMV's input language has no support for inlining portions of other files, but it does provide the `DEFINE` directive for establishing macro definitions. These directives are exploited in encoding predicates into model checker input.

For each defined predicate, the SPIN translator will output a `#define` to define the name of the predicate as a disjunction of Promela location predicates (`name[pid]@label` is true if the process with type `name` and id `pid` is currently executing at the statement following `label`). For each embedded predicate, the SPIN translator introduces a `bit` variable and appropriate transitions to set and reset it. Selected definitions for the `rw21` example are illustrated in Figure 7. The analogous definitions produced by the SMV translator are given in Figure 8. We note that for SMV, a predicate's boolean state variable is declared next to the state variable for the task that the predicate is associated with. For TRANS inputs SMV uses the order of variable declaration in the file as the variable order in the BDD, so the placement of the predicate state variables is important for performance.

The predicate names can be used directly in LTL formulae fed to the `spin -f` or in CTL formulae appended to the SMV transition system input.

The SPIN translator allows an additional option which will insert a directive for the model checker to incorporate the Büchi automaton for the LTL specification to be checked. For example, the command:

```
USER > (spin-input :never "prop4a")
```

will cause the SPIN translator to add the line

```
#include "prop4a.never"
```

```
  callstartwrite : boolean;
  callstopwrite : boolean;
  callwriteWork : boolean;

DEFINE
-- (IN-TASK READ_WRITE_CONTROL (= ACTIVEREADERS 0))
  activereaders0 := ( ( read__write__control = s11 )
      | ( read__write__control = s10 )
      | ( read__write__control = s9 )
      | ( read__write__control = s8 )
      | ( read__write__control = s6 )
      | ( read__write__control = s3 )
      | ( read__write__control = s2 )
      | ( read__write__control = s1 ) );
```

Figure 7: Embedded and Defined Predicates for SMV

at the bottom of the Promela file. If the `:never` parameter contains a '.', it will be used as-is (.never will not be appended). Note that checking deadlock is built into SPIN and requires no specification.

Unfortunately, the SMV input language does not support a `#include`-like directive. Currently, the `.trans` file are edited to append the CTL specification. A simple strategy for incorporating different CTL specifications into a transition system definition is to simply concatenate the files and submit them to SMV, e.g.:

```
$ cat sys.trans property.ctl | smv
```

Checking for deadlock with CTL requires a specification that is dependent on the final states of the CFSA. The `smv-input` command will automatically generate this specification when given the `:spec :deadlock` option.

10 Interpreting Counter-examples (step 7)

One of the nice features of the predicate definition mechanism is that counter-examples for failed model checks are presented, e.g., by SMV, in terms of changes in the values of state variables and `DEFINE`d quantities. Since defined and embedded predicates use these mechanisms, this makes it very easy to interpret the counter-example in terms of the property and a source-level model of system behavior.

For example, the property:

```
AG(callstartwrite -> AF(callstartread1 | callstartread2))
```

fails to hold on the `rw21` system. SMV produces a counter-example (given on the web-site), that gives the changes in state variable and `DEFINE` values along a path through the system on which the property does not hold. Eliminating all information in the counter-example except the boolean variables for the embedded predicates and the `DEFINE`s for the defined predicates that appear in the property specification gives the counter-example in Figure 8. It is easy to see from this counter-

```
-- specification AG (callstartwrite -> AF (callstartread1... is false
-- as demonstrated by the following execution sequence
state 1.1:
callstartwrite = 0
callstartread2 = 0
callstartread1 = 0

state 1.2:
callstartwrite = 1

state 1.3:
callstartwrite = 0

-- loop starts here --
state 1.5:
callstartwrite = 1

state 1.6:
callstartwrite = 0
```

Figure 8: Reduced SMV Counter-example

example that the loop consists solely of task rendezvous as a result of calls to the `Start_Write` entries of the `Read_Write_Control` task[5]. In other words, if the writer continually requests and is granted service by the controller then the readers will never start. Without predicate definitions interpretation of this counter-example is significantly more complicated (e.g., one needs to compute the set of predicates that are true in each state).

For embedded predicates, SPIN counter-examples provide essentially the same information as with SMV. Further work is required to provide similar support for defined predicates in SPIN.

11 Extensions to the Translation Tools

There are several potential improvements that might be made to these tools.

- Some model checkers have multiple input formats. Some of these formats may result in non-trivial differences in the time required for the model checker to process its input. For example, using transition systems with variables for tasks (as in an earlier version of the INCA tools) may yield significantly reduced SPIN translation times.
- Input parameters to entry calls are supported by the predicate definition mechanism. This should be generalized to support the partial specification of input parameters (e.g., specify the value of the first and third, but leave the second as *don't care*).

[5]States 1.4 and 1.7 of the counter-example were elided since they exhibit no state changes of the relevant predicates

- This general form of parameter modeling should be supported for procedure calls and returns as well.
- Global variables are not modeled. Many concurrent programs use some shared data (e.g., static class data in Java) that is not associated with a specific task. Support for this will be required.
- Description of global relational predicates could potentially be beneficial.

12 Conclusion

We have described the capabilities and illustrated the usage of a toolset that supports a seven step methodology for model checking properties of an Ada program. Those steps are:

1. Conversion of the original Ada program to a safe finite-state variant
2. Translation to SEDL
3. Definition of program features to be reasoned about
4. Translation to model checker input
5. Property specification using defined features
6. Execution of the model checker
7. Interpretation of counter-examples for false results

We continue apply and enhance this toolset and will update this tutorial as appropriate.

References

[ABC+91] G.S. Avrunin, U.A. Buy, J.C. Corbett, L.K. Dillon, and J.C. Wileden. Automated analysis of concurrent systems with the constrained expression toolset. *IEEE Transactions on Software Engineering*, 17(11):1204–1222, November 1991.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[CGL94] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[Cha96] A.T. Chamillard. *An Empirical Comparison of Static Concurrency Analysis Techniques*. PhD thesis, University of Massachusetts at Amherst, May 1996.

[Cor93] James C. Corbett. The Inca specification language. Technical Report ICS-TR-93-06, University of Hawaii, Department of Information and Computer Science, 1993.

[Cor96] J.C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3), March 1996.

[Cor98] J.C. Corbett. Constructing compact models of concurrent java programs. *Software Engineering Notes*, 23(2):1–10, March 1998. Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis.

[DAC98a] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Patterns in property specifications for finite-state verification. Technical Report 98-9, Kansas State University, Department of Computing and Information Sciences, 1998.

[DAC98b] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Property specification patterns for finite-state verification. In Mark Ardis, editor, *Proceedings of the Second Workshop on Formal Methods in Software Practice*, pages 7–15, March 1998.

[Dam96] D.R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Eindhoven University of Technology, May 1996.

[DHN98] Matthew B. Dwyer, John Hatcliff, and Muhammad Nanda. Using partial evaluation to enable verification of concurrent software. *ACM Computing Surveys*, 1998. to appear.

[DP98a] Matthew B. Dwyer and Corina S. Pasareanu. Filter-based model checking of partial systems. In *Proceedings of the Sixth ACM SIGSOFT Symposium on Foundations of Software Engineering*, November 1998. to appear.

[DP98b] Matthew B. Dwyer and Corina S. Pasareanu. Model checking generic container implementations. Technical Report 98-10, Kansas State University, Department of Computing and Information Sciences, 1998.

[FSZ88] K. Forester, I. Shy, and S. Zeil. IRIS-Ada: An Arcadia Perspective. Arcadia Technical Report UCI–88–01, University of California, Irvine, 1988.

[HDL98a] John Hatcliff, Matthew B. Dwyer, and Shawn Laubach. Staging static analysis using abstraction-based program specialization. In *Proceedings of the 10th Programming Languages, Implementations, Logics and Programs Conference*, September 1998.

[HDL+98b] John Hatcliff, Matthew B. Dwyer, Shawn Laubach, Jason Mayans, and Nanda Muhammad. Automatically specializing software for finite-state verification. Technical Report 98-4, Kansas State University, Department of Computing and Information Sciences, 1998.

[Hol97] G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.

[LGS+95] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajiani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–44, 1995.

[McM93] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[MP91] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1991.