SRI International

April 29, 2004

SAL: Tutorial

Leonardo de Moura

Computer Science Laboratory • 333 Ravenswood Ave. • Menlo Park, CA 94025 • (650) 326-6200 • Facsimile: (650) 859-2844

Contents

Chapter 1

Introduction

The SAL environment (SALenv) provides an integrated environment and a set of batch commands for the development and analysis of SAL specifications. This tutorial provides a short introduction to the usage of the main functionalities of SALenv.

1.1 The SAL environment

SALenv runs on PC systems running RedHat Linux or Windows XP (under cygwin), and Solaris (SPARC) workstations. We also believe that the system can be compiled for any UNIX variant. SALenv is implemented in Scheme and C++, but it is not necessary to know any of these languages to effectively use the system. Actually, basic notions of the Scheme language are necessary if you intend to customize the behavior of SALenv.

1.2 The SAL language

The SAL language is not that different from the input languages used by various other verification tools such as SMV, Murphi, Mocha, and SPIN. Like these languages, SAL describes transition systems in terms of initialization and transition commands. These can be given by variable-wise definitions in the style of SMV or as guarded commands in the style of Murphi.

1.3 Examples

In this tutorial we describe the SAL language by presenting some examples. A complete description of the SAL language is available online at `http://sal.csl.sri.com/`. The abstract syntaxt tree of SAL is specified in XML. The SALenv tools also accept two machine-friendly formats for other programs: XML and a Lisp-like syntax.

1.4 SALPATH

All SALenv tools search for context files in the path specified by the environment variable `SALPATH`. If you are using `bash`, then you should modify your `.bashrc` file. For instance, to set `SALPATH` to the current and the `/homes/leonardo/tmp` directories, you should add the following command to your `.bashrc` file:

```
export SALPATH=.:/homes/leonardo/tmp
```

If you are using `tcsh`, then you should modify your `.cshrc` file:

```
setenv SALPATH .:/homes/leonardo/tmp
```

If the environment variable `SALPATH` is not specified, then the tools will only search for SAL context files in the current directory.

Chapter 2

A Simple Example

Consider the simple SAL specification in Figure 1. A SAL context is a "container" of types, constants, and modules (i.e., transition systems). The specification in Figure 1 specifies the context `short`, which contains a type (`State`), and a module declaration (`main`). The type `State` is an enumerated type consisting of two elements: `ready`, and `busy`. The module `main` specifies a transition system which contains: a boolean input variable (`request`), and an output variable (`state`) of type `State`. The variable `state` is initialized with the value `ready`. The `TRANSITION` section specifies a "step" of the transition system. In SAL, the notation `X'` is used to denote the next value of the variable `X`. In the module `main`, the next value of `state` is `busy` when the current value is `ready` and the input `request` is true, otherwise the next value is any element in the set `{ready, busy}` (remark: the `IN` construct is used to specify nondeterministic assignments). Observe that a module can only specify the initial and next values of controlled variables: output, local and global variables.

```
short: CONTEXT =                                                    1
BEGIN
  State: TYPE = {ready, busy};

  main: MODULE =
    BEGIN
      INPUT request : BOOLEAN
      OUTPUT state : State
      INITIALIZATION
        state = ready
      TRANSITION
        state' IN IF (state = ready) AND request THEN
                    {busy}
                  ELSE
                    {ready, busy}
                  ENDIF
    END;
END
```

The context `short` is available on the SAL distribution package in the `examples` directory.

2.1 Simulator

SALenv contains a simulator for finite state specifications based on BDDs (Binary Decision Diagrams). The simulator allows users to explore different execution paths of a SAL specification. In this way, users can increase their confidence in their model before performing verification. Actually, the SALenv simulator is not a regular simulator, since it is scriptable, that is, users can use the Scheme programming language to implement new simulation and verification procedures, and to automate repetitive tasks.

Now, assuming that the current directory contains the file `short.sal`, the current directory is in the `SALPATH`, and the SALenv tools are in the `PATH`, the simulator can be started executing the command `sal-sim`. The symbol `%` is used to represent the system prompt.

```
% sal-sim
SAL Simulator (Version 2.1). Copyright (c) 2003, 2004 SRI International.
Build date: Wed Jan 21 11:34:54 PST 2004
Type ‘(exit)’ to exit.
Type ‘(help)’ for help.

sal >
```

Now, users can import the context `short` using the command:

```
sal > (import! "short")
```

Command `(help-commands)` prints the main commands available in the simulator. The following commmand can be used to start the simulation of the module `main`.

```
sal > (start-simulation! "main")
```

The simulation initialized by command `start-simulation!` is composed of: a current trace, a current finite state machine, and a set of already visited states. Actually, the “current trace” may represent a set of traces, since a trace is list of set of states. Command `(display-curr-trace)` prints one of the traces in the “current trace”. Initially, the “current trace” contains just the set of initial states. We say the first element in the list “current trace” is the set of current states. Command `(display-curr-states)` displays the set of current states, its default behavior is to print at most 10 states, but (display-curr-states <num>) can be used to print at most <num> states. Command `(display-curr-states)` assigns an index (positive number) to the printed states. Then, users can use this index to peek at a specific state using the command (select-state! <idx>), which restricts the set of current states to the single selected state.

```
sal > (display-curr-states)
State 1
--- Input Variables (assignments) ---
(= request true);
--- System Variables (assignments) ---
(= state ready);
-----------------------------
State 2
--- Input Variables (assignments) ---
(= request false);
--- System Variables (assignments) ---
(= state ready);
-----------------------------

sal > (select-state! 1)

sal > (display-curr-states)
State 1
--- Input Variables (assignments) ---
(= request true);
--- System Variables (assignments) ---
(= state ready);
-----------------------------
```

Command `(step!)` performs a simulation step, that is, it appends the successors of the set of current states in the current trace. Clearly, the set of current states is also updated.

```
sal > (step!)

sal > (display-curr-trace)
Step 0:
--- Input Variables (assignments) ---
(= request true);
--- System Variables (assignments) ---
(= state ready);
------------------------
Step 1:
--- Input Variables (assignments) ---
(= request false);
--- System Variables (assignments) ---
(= state busy);
```

Command `(filter-curr-states! <constraint>)` provides an alternative way to select a subset of the set of current states. The argument of `filter-curr-states!` is a BDD or a SAL expression. The new set of current states will contain only states that satisfy the given constraint.

```
sal > (filter-curr-states! "(not request)")

sal > (display-curr-states)
State 1
--- Input Variables (assignments) ---
(= request false);
--- System Variables (assignments) ---
(= state busy);
-----------------------------
```

As described before, users can automate repetitive tasks using the Scheme programming language. For instance, the following example shows how to define a new command called `(n-step! n)`:

```
sal > (define (n-step! n)
        (when (> n 0)
          (select-state! 1)
          (step!)
          (n-step! (- n 1))))

sal > (n-step! 3)

sal > (display-curr-trace)
Step 0:
--- Input Variables (assignments) ---
(= request true);
--- System Variables (assignments) ---
(= state ready);
------------------------
...
------------------------
Step 4:
--- Input Variables (assignments) ---
(= request true);
--- System Variables (assignments) ---
(= state ready);
```

User defined commands such as `(n-step! n)` can be stored in files and loaded in the simulator using command `(load "<file-name>")`.

Command `(sal/reset!)` forces garbage collection and reinitializes all datastructures (e.g., caches) used by the simulator. It is useful to call `(sal/reset!)` before starting the simulation of a different module.

2.2 Path Finder

The `sal-path-finder` is a random trace generator for SAL modules based on SAT solving. For instance, the following command produces a trace for the module `main` located in the context `short`.

```
% sal-path-finder short main
========================
Path
========================
Step 0:
--- Input Variables (assignments) ---
request = false;
--- System Variables (assignments) ---
state = ready;
------------------------
...
------------------------
Step 10:
--- Input Variables (assignments) ---
request = false;
--- System Variables (assignments) ---
state = ready;
```

The default behavior of `sal-path-finder` is to produce a trace with 10 transitions. The option `--depth=<num>` can be used to control the length of the trace.

```
% sal-path-finder --depth=5 short main
...
```

2.3 Model Checking

SALenv contains a symbolic model checker called `sal-smc`. `sal-smc` allows users to specify properties in linear temporal logic (LTL), and computation tree logic (CTL). However, in the current version SALenv does not print counterexamples for CTL properties. When users specify an invalid property in LTL, a counterexample is produced. LTL formulas state properties about each linear path induced by a module (transition system). Typical LTL operators are:

- `G(p)` (read "always p"), stating that `p` is always true.
- `F(p)` (read "eventually p"), stating that `p` will be eventually true.
- `U(p,q)` (read "p until q"), stating that `p` holds until a state is reached where `q` holds.
- `X(p)` (read "next p"), stating that `p` is true in the next state.

For instance, the formula `G(p => F(q))` states that whenever `p` holds, `q` will eventually hold. The formula `G(F(p))` states that `p` holds infinitely often.

Typical CTL operators are:

- `AG(p)`, stating that `p` is globally true.
- `EG(p)`, stating that there is a path where `p` is continuously true.
- `AF(p)`, stating that for all paths `p` is eventually true.

- `EF(p)`, stating that there is a path where `p` is eventually true.
- `AU(p,q)`, stating that in all paths `p` holds until a state is reached where `q` holds.
- `EU(p,q)`, stating that there is a path where `p` holds until a state is reached where `q` holds.
- `AX(p)`, stating that `p` holds in all successor states.
- `EX(p)`, stating that there is a successor state where `p` holds.

Figure 2 contains three different ways to state the same property of the module `main`. The third property uses the `ltllib` context, which defines several "macros" for commonly used LTL patterns. `ltllib!responds_to` is a qualified name in SAL, it is a reference to the function `responds_to` located in the context `ltllib`.

```
th1: THEOREM main |- AG(request => AF(state = busy));

th2: THEOREM main |- G(request => F(state = busy));

th3: THEOREM main |- ltllib!responds_to(state = busy, request);
```

2

These properties can be verified using the following commands:

```
% sal-smc short th1
proved.
% sal-smc short th2
proved.
% sal-smc short th3
proved.
```

SALenv also contains a bounded model checker called `sal-bmc`. This model checker only supports LTL formulas, and it is basically used for refutation, although it can produce proofs by induction for safety properties.

```
% sal-bmc short th2
no counterexample between depths: [0, 10].
% sal-bmc short th3
no counterexample between depths: [0, 10].
```

The default behavior is to look for counterexamples up to depth 10. The option `--depth=<num>` can be used to control the depth of the search. The option `--iterative` forces the model checker to use iterative deepening, and it is useful to find the shortest counterexample for a given property.

```
% sal-bmc --depth=20 short th2
no counterexample between depths: [0, 20].
```

Chapter 3

The Peterson Protocol

In this chapter, we illustrate SAL model checking via a simplified version of Peterson's algorithm for 2-process mutual exclusion. The SAL files for this example are located in the following subdirectory in the SAL distribution package: `examples/peterson`. The 2-process version of the mutual exclusion problem requires that two processes are never simultaneously in their respective critical sections. The behavior of each process is modeled by a SAL module. Actually, we use a parametric SAL module to specify the behavior of both processes. The prefix `pc` denotes *program counter*. When `pc1` (`pc2`) is set to the value `critical`, process 1(2) is in its critical section. The noncritical section has two self-explanatory phases: `sleeping` and `trying`. Each process is allowed to observe whether or not the other process is `sleeping`. The variables `x1` and `x2` control the access to the critical section.

Figure 3 contains the specification of the context `peterson`. `PC` is an enumerated type. This type consists of three values: `sleeping`, `trying`, and `critical`. Since the behavior of the two processes in the Peterson's protocol is quite similar, a parametric SAL module (`mutex`) is used to specify them. In this way, `process[FALSE]` describes the behavior of the first process, and `process[TRUE]` the behavior of the other one. It is important to note that the variable `pc1` in the module `process` represents the program counter of the current process, and `pc2` the program counter of the other process. It is a good idea to label guarded commands, since it helps us to understand the counterexamples. So, the following labels are used: `wakening`, `entering_critical`, and `leaving_critical`.

```
peterson: CONTEXT =                                                    3
BEGIN
  PC: TYPE = sleeping, trying, critical;

  process [tval : BOOLEAN]: MODULE =
    BEGIN
      INPUT pc2 : PC
      INPUT x2 : BOOLEAN
      OUTPUT pc1 : PC
      OUTPUT x1 : BOOLEAN
      INITIALIZATION
        pc1 = sleeping
      TRANSITION
        [
         wakening:
           pc1 = sleeping --> pc1' = trying; x1' = x2 = tval
         []
         entering_critical:
           pc1 = trying AND (pc2 = sleeping OR x1 = (x2 /= tval))
           --> pc1' = critical
         []
         leaving_critical:
           pc1 = critical --> pc1' = sleeping; x1' = x2 = tval
        ]
    END;

END
```

Initially, the program counter is set to `sleeping`. The transition section is composed by three guarded commands which describe the three phases of the algorithm. The entire system is specified by performing the asynchronous composition of two instances of the module `process`.

```
  system: MODULE =
     process[FALSE]
     []
     RENAME pc2 TO pc1, pc1 TO pc2,
            x2 TO x1, x1 TO x2
     IN process[TRUE];
```

3.1 Path Finder

The following command can be used to obtain an execution trace (with 5 steps) of the Peterson's protocol.

```
% sal-path-finder -v 2 -d 5 peterson system
...
```

The option `-v 2` sets the verbosity level to 2, the produced verbose messages allow users to follow the steps performed by the SALenv tools. The option `-d 5` sets the number of execution steps. Figure 4 contains a fragment of the trace produced by `sal-path-finder`. The trace contains detailed information about each transition performed.

```
Step 0:                                                                    4
--- System Variables (assignments) ---
pc1 = sleeping;
pc2 = sleeping;
x1 = false;
x2 = false;
------------------------
Transition Information:
(module instance at [Context: scratch, line(1), column(11)]
  (module instance at [Context: peterson, line(33), column(10)]
    (label wakening
      transition at [Context: peterson, line(14), column(10)])))
------------------------
Step 1:
--- System Variables (assignments) ---
pc1 = sleeping;
pc2 = trying;
x1 = false;
x2 = false;
------------------------
...
```

The following command uses ZCHAFF to obtain an execution trace (with 20 steps) of the Peterson's protocol. You must have ZCHAFF installed in your machine to use this command. ZCHAFF is *not* part of the SAL distribution package.

```
% sal-path-finder -v 2 -d 20 -s zchaff peterson system
...
```

3.2 Model Checking

The main property of the Peterson's protocol is mutual-exclusion, that is, it is not possible for more than one process to enter the critical section at the same time. This safety property can be stated in the following way:

```
  mutex: THEOREM system |- G(NOT(pc1 = critical AND pc2 = critical));
```

The following command can be used to prove this property.

```
% sal-smc -v 3 peterson mutex
```

In `sal-smc`, the default proof method for safety properties is *forward reachability*. The option `backward` can be used to force `sal-smc` to perform *backward reachability*.

```
% sal-smc -v 3 --backward peterson mutex
...
proved.
```

In this example, backward reachability needs fewer iterations to reach the fix point.

This property can also be proved using k-induction (option `-i` in `sal-bmc`). Actually 2-induction is sufficient to prove this property.

```
% sal-bmc -v 3 -d 2 -i peterson mutex
...
proved.
```

It is important to note that there are several trivial algorithms that satisfy the mutual exclusion property. For instance, an algorithm that all jobs do not perform any transition. Therefore, it is important to prove liveness properties. For instance, we can try to prove that every process reach the critical section infinitely often. The following LTL formula states this property:

```
  livenessbug1: THEOREM system |- G(F(pc1 = critical));
```

Before proving a liveness property, we must check if the transition relation is total, that is, if every state has at least one successor. The model checkers may produce *unsound* results when the transition relation is not total. The totality property can be verified using the `sal-deadlock-checker`.

```
% sal-deadlock-checker -v 3 peterson system
...
ok (module does NOT contain deadlock states).
```

Now, we use `sal-smc` to check the property `livenessbug1`.

```
% sal-smc -v 3 peterson livenessbug1
...
Step 0:
...
========================
Begin of Cycle
========================
...
```

Unfortunately, this property is not true. A counterexample for a LTL liveness property is always composed of a prefix, and a cycle. For instance, the counterexample for the property `livenessbug1` describes a cycle where the process 2 does not perform a transition.

There is not guarantee that `sal-smc` will produce the shortest counterexample for a liveness property. However, it is possible to use `sal-bmc` to produce the shortest counterexample.

```
% sal-bmc -v 3 -it peterson livenessbug1
...
Counterexample:
...
```

It is important to note that `sal-bmc` is usually more efficient for counterexample detection.

Since, `livenessbug1` is not a valid property, we can try to prove the weaker liveness property:

```
  liveness1: THEOREM system |- G(pc1 = trying => F(pc1 = critical));
```

This property states that if process 1 is trying to enter the critical section, it will eventually succeed. The following command can be used to prove the property:

```
% sal-smc -v 3 peterson liveness1
...
proved.
```

Chapter 4

The Bakery Protocol

In this chapter, we specify the bakery protocol. The SAL files for this example are located in the following subdirectory in the SAL distribution package: `examples/bakery`. The basic idea is that of a bakery, where customers (jobs) take numbers, and whoever has the lowest number gets service next. Here, of course, "service" means entry to the critical section. The version of the bakery protocol described in this chapter is finite state, since we want to model-check it using `sal-smc`, and `sal-bmc`. So, in our version there is a maximum "ticket" value. Figure 5 contains the the header of the context `bakery`, and the type declarations. The context `bakery` has two parameters: `N` is the number of (potential) customers, and `B` is the maximum ticket value. Both values must be non-zero natural numbers. The type `Job_Idx` is a subrange that is used to identify the customers. The type `Ticket_Idx` is also a subrange, where `0` represents the "null" ticket. The type of the next ticket to be issued (`Next_Ticket_Idx`) is also a subrange, where `B+1` represents the "no ticket available" condition. The type of the "resources" of the system (`RSRC`) is a record with two fields: `data`, an array which stores the "ticket" of each job (customer); `next-ticket`, the value of the next ticket to be issued. We say the system is *saturated*, when the field `next_ticket` is equals to `B+1`. Each job (customer) has a control variable of type `Job_PC`, an enumerated type consisting of the three values: `sleeping`, `trying`, and `critical`.

```
bakery{; N : nznat, B : nznat}: CONTEXT =                             5
BEGIN
  Job_Idx: TYPE = [1..N];

  Ticket_Idx: TYPE = [0..B];

  Next_Ticket_Idx: TYPE = [1..(B + 1)];

  RSRC: TYPE =  [# data: ARRAY Job_Idx OF Ticket_Idx,
                   next_ticket: Next_Ticket_Idx #];

  Job_PC: TYPE = {sleeping, trying, critical};

...

END
```

Figure 6 contains auxiliary functions used to specify the bakery protocol. Function `min_non_zero_ticket` returns the "ticket" of the job (customer) to be "served", the possible return values are:

- `0` when there is no job (customer) with a non-zero ticket (no customer condition).
- `n > 0`, where `n` is the minimal (non-zero) ticket issued to a job (customer).

The auxiliary (recursive) function `min_non_zero_ticket_aux` is used to traverse the array `rsrc.data`. The function `min` is a builtin function that returns the minimum of two numbers. The function `can_enter_critical?` returns true, when `job_idx` can enter the critical section by comparing the customer's ticket with the value returned by `min_non_zero_ticket`. The function `next_ticket` issues a new ticket to the job `job_idx`, that is, it updates the array `rsrc.data` at position `job_idx`, and increments the counter `rsrs.next_ticket`. In SAL, expressions do not have side-effects. For instance, the update expression `x WITH [idx] := v` results in an array that is equals to `x`, except that at position `idx` it takes the value `v`. The function `reset_job_ticket` assigns the "null ticket" to the job `job_idx`. The function `can_reset_ticket_counter` returns true, when it is safe to reset the `rsrc.next_ticket` counter.

```
min_non_zero_ticket_aux(rsrc : RSRC, idx : Job_Idx) : Ticket_Idx =        6
 IF idx = N THEN rsrc.data[idx]
 ELSE LET curr: Ticket_Idx = rsrc.data[idx],
          rest: Ticket_Idx = min_non_zero_ticket_aux(rsrc, idx + 1)
      IN IF curr = 0 THEN rest
         ELSIF rest = 0 THEN curr
         ELSE min(curr, rest)
         ENDIF
 ENDIF;

min_non_zero_ticket(rsrc : RSRC) : Ticket_Idx =
  min_non_zero_ticket_aux(rsrc, 1);

can_enter_critical?(rsrc : RSRC, job_idx : Job_Idx): BOOLEAN =
  LET min_ticket: Ticket_Idx = min_non_zero_ticket(rsrc),
      job_ticket: Ticket_Idx = rsrc.data[job_idx]
  IN job_ticket <= min_ticket;

saturated?(rsrc : RSRC): BOOLEAN =
  rsrc.next_ticket = B + 1;

next_ticket(rsrc : RSRC, job_idx : Job_Idx): RSRC =
  IF saturated?(rsrc) THEN rsrc
  ELSE (rsrc WITH .data[job_idx] := rsrc.next_ticket)
             WITH .next_ticket := rsrc.next_ticket + 1
  ENDIF;

reset_job_ticket(rsrc : RSRC, job_idx : Job_Idx): RSRC =
  rsrc WITH .data[job_idx] := 0;

can_reset_ticket_counter?(rsrc : RSRC): BOOLEAN  =
  (FORALL (j : Job_Idx): rsrc.data[j] = 0);

reset_ticket_counter(rsrc : RSRC): RSRC =
  rsrc WITH .next_ticket := 1;
```

Since the behavior of each job (customer) is almost identical, we use a parametric SAL module to specify them (Figure 7). In this way, `job[1]` denotes the first job, `job[2]` the second, and so on. The local variable `pc` contains the program counter of a job, and it is initialized with the value `sleeping`. The global variable `rsrc` contains the shared "resources" of the system. The transition section is specified using three labeled guarded commands: `wakening`, `entering_critical_section`, and `leaving_critical_section`. Labeled commands are particularly useful in the generation of readable counterexamples. A guarded command is composed of a guard, and a sequence of assignments. The guard is a boolean expression, and a guarded command is said to be *ready to execute* when the guard is true. If more than one guarded command is ready to execute, a nondeterministic choice is performed. For instance, the guarded command `wakening` is ready to execute, when the current value of `pc` is `sleeping`, and the system is not "saturated". If the next value of a controlled (local, output, and global) variable `x` is not specified by a guarded command, then `x` maintains its current value, that is, the guarded command contains an "implicit" assignment `x' = x`. For instance, in the guarded command `entering_critical_section` the variable `rsrc` is not modified.

```
job [job_idx : Job_Idx]: MODULE =                                          7
  BEGIN
    GLOBAL rsrc : RSRC
    LOCAL pc : Job_PC
    INITIALIZATION
      pc = sleeping
    TRANSITION
      [
       wakening:
         pc = sleeping AND NOT(saturated?(rsrc))
         --> pc' = trying;
             rsrc' = next_ticket(rsrc, job_idx)
       []
       entering_critical_section:
         pc = trying AND can_enter_critical?(rsrc, job_idx)
         --> pc' = critical
       []
       leaving_critical_section:
         pc = critical --> pc' = sleeping;
                           rsrc' = reset_job_ticket(rsrc, job_idx)
      ]
  END;
```

Figure 8 specifies an auxiliary module that is used to initialize the shared variable `rsrc`, and to reset the next-ticket counter when the system is "saturated". Note that the array literal `[[j : Job_Idx] 0]` is used to initialize the field `data`.

```
controller: MODULE =                                                       8
  BEGIN
    GLOBAL rsrc : RSRC
    INITIALIZATION
      rsrc = (# data := [[j : Job_Idx] 0], next_ticket := 1 #)
    TRANSITION
      [
        reseting_ticket_counter:
          can_reset_ticket_counter?(rsrc)
          --> rsrc' = reset_ticket_counter(rsrc)
      ]
  END;
```

The whole system is obtained by composing `N` instances of the module `job`, and one instance of the module `controller` (Figure 9). The auxiliary module `jobs` is the multi-asynchronous composition of `N` instances of `job`, since the type `Job_Idx` is a subrange `[1..N]`. Notice that each instance of `job` is initialized with a different index. In a multi-asynchronous (and multi-synchronous) composition, all local variables are implicitly mapped to arrays. For instance, the local variable `pc` of each `job` instance is implicitly mapped to pc[job_idx], where the type of pc in the module jobs is ARRAY Job_Idx OF Job_PC. This kind of mapping is necessary, since users may need to reference the local variables of different instances when specifying a property. The module system is the asynchronous composition of the modules controller and jobs.

```
jobs : MODULE = ([] (job_idx : Job_Idx): job[job_idx]);                9

system: MODULE = controller [] jobs;
```

The SAL context bakery is available in the SAL distribution package in the examples directory.

4.1 Path Finder

The following command can be used to obtain a trace of an instance of the bakery protocol with 3 jobs, and maximum ticket number equals to 7.

```
% sal-path-finder -v 2 --depth=5 --module='bakery{;3,7}!system'
...
```

The option -v 2 sets the verbosity level to 2, the produced verbose messages allow users to follow the steps performed by the SAL tools. The module to be simulated is specified using the option --module because the context bakery is parametric.

Figure 10 contains a fragment of the trace produced by sal-path-finder. The trace contains detailed information about each transition performed. For instance, the first transition was performed by the guarded command wakening of job 3 (job_idx = 3).

10

```
Step 0:
--- System Variables (assignments) ---
pc[1] = sleeping;
pc[2] = sleeping;
pc[3] = sleeping;
rsrc.data[1] = 0;
rsrc.data[2] = 0;
rsrc.data[3] = 0;
rsrc.next-ticket = 1;
------------------------
Transition Information:
(module instance at [Context: scratch, line(1), column(11)]
  (module instance at [Context: bakery, line(116), column(8)]
    (label reseting-ticket-counter
      transition at [Context: bakery, line(111), column(10)])))
------------------------
Step 1:
--- System Variables (assignments) ---
pc[1] = sleeping;
pc[2] = sleeping;
pc[3] = sleeping;
rsrc.data[1] = 0;
rsrc.data[2] = 0;
rsrc.data[3] = 0;
rsrc.next-ticket = 1;
------------------------
...
```

4.2 Model Checking

The main property of the bakery protocol is mutual-exclusion, that is, it is not possible for more than one job to enter the critical section at the same time. This safety property can be stated in the following way:

```
  mutex: THEOREM
    system |- G(NOT (EXISTS (i : Job_Idx, j : Job_Idx):
                       i /= j AND
                       pc[i] = critical AND
                       pc[j] = critical));
```

The following command can be used to prove this property for 5 customers and maximum ticket value equals to 15.

```
% sal-smc -v 3 --assertion='bakery{;5,15}!mutex'
...
proved.
```

The assertion to be verified is specified using the option `--assertion` because the context `bakery` is parametric. In `sal-smc`, the default proof method for safety properties is *for-*

ward reachability. The option backward can be used to force sal-smc to perform *backward reachability*.

```
% sal-smc -v 3 --backward --assertion='bakery{;5,15}!mutex'
...
proved.
```

In this example, backward reachability needs fewer iterations to reach the fix point, but it is less efficient, and consumes much more memory than forward reachability.

The default behavior of sal-smc is to build a partitioned transition relation composed of several BDDs. However, the option --monolithic forces sal-smc (and sal-sim) to build a monolithic (a single BDD) transition relation. The option --cluster-size=<num> controls the generation of clusters in a partitioned transition relation, the idea is that two clusters (BDDs) are only combined into a single cluster if their sizes are below the threshold.

```
% sal-smc -v 3 --monolithic --assertion='bakery{;5,15}!mutex'
...
% sal-smc -v 3 --max-cluster-size=32768
         --assertion='bakery{;5,15}!mutex'
...
```

In sal-smc the BDD variables are (re)ordered to minimize the size of the BDDs. Variable (re)ordering is performed in the following stages of sal-smc.

- First, an initial (static) variable order is built. The option --static-order=<name> sets the algorithm used to build the initial order.
- After the construction of the transition relation, one or more forced variable reordering may be performed. The default behavior is one forced variable reordering. The option --num-reorders=<num> sets the number of variable reorderings. The option -r <name> sets the reordering strategy (the default strategy is sift). Use the option --help to obtain the available strategies.
- Dynamic variable reordering is not enabled, but the option --enable-dynamic-reorder can be used to enable it. Dynamic variable reordering also uses the strategy specified by the option -r <name>.

It is important to note that there are several trivial algorithms that satisfy the mutual exclusion property. For instance, an algorithm that all jobs do not perform any transition. Therefore, it is important to prove liveness properties. For instance, we can try to prove that every process reaches the critical section infinitely often. The following LTL formula states this property:

```
  liveness_bug: THEOREM
    system |- (FORALL (i : Job_Idx): G(F(pc[i] = critical)));
```

Before proving a liveness property, we must check if the transition relation is total, that is, if every state has at least one successor. The model checkers may produce *unsound* results when the transition relation is not total. The totality property can be verified using the sal-deadlock-checker.

```
% sal-deadlock-checker -v 3 --module='bakery{;5,15}!system'
...
ok (module does NOT contain deadlock states).
```

Now, we use `sal-smc` to check the property `liveness_bug`.

```
% sal-smc -v 3 --assertion='bakery{;5,15}!liveness_bug'
...
Counterexample:
Step 0:
...
========================
Begin of Cycle
========================
...
```

Unfortunately, this property is not true. A counterexample for a LTL liveness property is always composed of a prefix, and a cycle. For instance, the counterexample for the property `liveness_bug` describes a cycle where at least one of the jobs do not perform a transition. A simpler counterexample can be produced if we try to verify an instance of the protocol with only two jobs.

```
% sal-smc -v 3 --assertion='bakery{;2,3}!liveness_bug'
...
Counterexample:
...
```

It is also possible to use `sal-bmc` to produce the shortest counterexample.

```
% sal-bmc -v 3
          --iterative
          --assertion='bakery{;5,15}!liveness_bug'
...
Counterexample:
...
```

In this example, `sal-bmc` finds the counterexample in less time. Actually, `sal-bmc` is usually more efficient for counterexample detection.

Since `liveness_bug` is not a valid property, we can try to prove the weaker liveness property:

```
  liveness: THEOREM
    system |- (FORALL (i : Job_Idx):
                 G(pc[i] = trying => F(pc[i] = critical)));
```

This property states that every job trying to enter the critical section will eventually succeed. The following command can be used to prove the property:

```
% sal-smc -v 3 --assertion='bakery{;5,15}!liveness'
...
proved.
```

Chapter 5

Synchronous Bus Arbiter

The synchronous bus arbiter is a classical example in symbolic model checking. The example described here was extracted from McMillan's doctoral thesis. The SAL files for this example are located in the following subdirectory in the SAL distribution package: `examples/arbiter`. respectively. The purpose of the arbiter is to grant access on each clock cycle to a single client among a number of clients contending for the use of a bus (or another resource). The inputs of the circuit are a set of request signals, and the output a set of acknowledge signals. Normally, the arbiter asserts the acknowledge signal to the client with lowest signal. However, as signals become more frequent, the arbiter is designed to fall back on round robin scheme, so that every requester is eventually granted access. This is done by circulating a token in a ring of arbiter cells, with one cell per client. The token moves once every clock cycle. If a given client's request persists for the time it takes for the token to make a complete circuit, that client is granted immediate access to the bus. Figure 11 contains the header of the context `arbiter`, and the type declarations. The context `arbiter` has a parameter `n` which is the number of clients. The type of `n` is a subtype of `NATURAL`, since `n` must be greater than 1.

```
arbiter{; n : {x : NATURAL | x > 1}}: CONTEXT =                    11
BEGIN

  Range: TYPE = [1..n];
  Array: TYPE = ARRAY Range OF BOOLEAN;

  ...
END
```

Figure 12 contains the specification of a basic cell of the arbiter. Each cell has a request input (`req`) and acknowledgement output (`ack`). The grant output (`grant_out`) of cell *i* is passed to cell *i+1*, and indicates that no clients of index less than or equal to *i* are requesting. Each cell has a local variable `t` which stores a true value when the cell has the token. The `t` local variables form a circular shift register which shifts up one place each clock cycle. Each cell also has a local variable `w` (*waiting*) which is set to true when the `req` is true and the token is present. The value of `w` remains true while the request persists, until the token returns. At this time, the cell's override (`override_out`) and acknowledgement (`ack`) outputs are set to true. The override signals propagates to the cells below, negating the grant input of cell 1, and thus preventing any other cells from acknowledging at the same time.

```
cell [initial_t : BOOLEAN]: MODULE =                                12
  BEGIN
    INPUT req : BOOLEAN
    INPUT token_in : BOOLEAN
    INPUT override_in : BOOLEAN
    INPUT grant_in : BOOLEAN
    OUTPUT ack : BOOLEAN
    OUTPUT token_out : BOOLEAN
    OUTPUT override_out : BOOLEAN
    OUTPUT grant_out : BOOLEAN
    LOCAL t : BOOLEAN
    LOCAL w : BOOLEAN
    LOCAL aux : BOOLEAN
    DEFINITION
      token_out = t;
      aux = w AND t;
      override_out = override_in OR aux;
      grant_out = grant_in AND NOT(req);
      ack = req AND (aux OR grant_in)
    INITIALIZATION
      w = FALSE;
      t = initial_t
    TRANSITION
      t' = token_in;
      w' = req AND (w OR t)
  END;
```

Figure 13 describes the composition of `n` instances of the module `cell`. The auxiliary module `aux_module` is used to provide a constant false value for the input variable `override_in` of the first cell. It is also used to "connect" the negation of the output `override_out` to the input `grant_in` in the last cell. The outputs and inputs of each cells are mapped to arrays using the construct `RENAME`. The auxiliary arrays are declared using the `WITH` construct. The construct `||` is the synchronous composition operator. So, the full arbiter is the synchronous composition of the auxiliary module, a cell, and the multisynchronous composition of `n-1` cells, since the type of `idx` is the subrange `[2..n]`.

13

```
aux_module : MODULE =
BEGIN
  OUTPUT zero_const : BOOLEAN
  INPUT aux : BOOLEAN
  OUTPUT inv_aux : BOOLEAN
  DEFINITION
    zero_const = FALSE;
    inv_aux = NOT(aux)
END;

arbiter: MODULE =
  WITH OUTPUT Ack : Array;
       INPUT Req : Array;
       OUTPUT Token : Array;
       OUTPUT Grant : Array;
       OUTPUT Override : Array
    (RENAME aux TO Override[n], inv_aux TO Grant[n]
     IN aux_module)
    ||
    (WITH INPUT zero_const : BOOLEAN
          (RENAME ack TO Ack[1],
                  req TO Req[1],
                  token_in TO Token[1],
                  token_out TO Token[n],
                  override_in TO zero_const,
                  override_out TO Override[1],
                  grant_in TO Grant[1]
           IN (LOCAL grant_out
               IN cell[TRUE])))
    ||
    (|| (idx : [2..n]):
          (RENAME ack TO Ack[idx],
                  req TO Req[idx],
                  token_in TO Token[idx],
                  token_out TO Token[idx - 1],
                  override_in TO Override[idx - 1],
                  override_out TO Override[idx],
                  grant_in TO Grant[idx],
                  grant_out TO Grant[idx - 1]
           IN cell[FALSE]));
```

The SAL context `arbiter` is available in the SAL distribution package in the `examples` directory.

5.1 Model Checking

The desired properties of the arbiter circuit are:

- No two acknowledge outputs are true simultaneously.

```
at_most_one_ack:
  THEOREM arbiter |- G((FORALL (i : [1..n - 1]):
                          (FORALL (j : [i + 1..n]):
                            NOT(Ack[i] AND Ack[j]))));
```

- Every persistent request is eventually acknowledged.

```
every_request_is_eventually_acknowledged:
  THEOREM arbiter |- (FORALL (i : [1..n]):
                       G(F(Req[i] => Ack[i])));
```

- Acknowledge is not true without a request.

```
no_ack_without_request:
  THEOREM arbiter |- G((FORALL (i : [1..n]): Ack[i] => Req[i]));
```

The following commands can be used to prove the properties for an arbiter with 30 cells:

```
% sal-smc -v 3 --assertion='arbiter{;30}!at_most_one_ack'
...
proved.
% sal-smc -v 3 --assertion='arbiter{;30}!every_request_is_eventually_acknowledged'
...
proved.
% sal-smc -v 3 --assertion='arbiter{;30}!no_ack_without_request'
...
proved.
```

The property `no_ack_without_request` can be proved using the *k-induction* rule in `sal-bmc`. This induction rule generalizes BMC in that it requires demonstrating the safety property p holds in the first k states of any execution, and that if p holds in every state of executions of length k, then every successor state also satisfies this invariant.

```
% sal-bmc -v 3 -i
          --assertion='arbiter{;30}!no_ack_without_request'
...
proved.
```

Although `at_most_one_ack` is a safety property, it is not feasible to prove it using induction, unless users provide auxiliary lemmas. The option `--display-induction-ce` can be used to force `sal-bmc` to display a counterexample for the inductive step.

```
% sal-bmc -v 3 -i -d 2 --display-induction-ce
          --assertion='arbiter{;3}!at_most_one_ack'
...
k-induction rule failed, please try to increase the depth.
Counterexample:
Step 0:
...
```

Inspecting the counterexample, you can notice that more than one cell has the token. So, we may use the following auxiliary lemma to prove the property `at_most_one_ack`.

```
  at_most_one_token:
    THEOREM arbiter |- G((FORALL (i : [1..n - 1]):
                           (FORALL (j : [i + 1..n]):
                            NOT(Token[i] AND Token[j]))));
```

The following command instructs `sal-bmc` to use `at_most_one_token` as an auxiliary lemma.

```
% sal-bmc -v 3 -i -d 1 --lemma=at_most_one_token
          --assertion='arbiter{;3}!at_most_one_ack'
...
proved.
```

It is important to observer that the previous proof is only valid if the property `at_most_one_token` is valid. The following command proves the auxiliary lemma `at_most_one_token` using 1-induction.

```
% sal-bmc -v 3 -i -d 1
           --assertion='arbiter{;30}!at_most_one_token'
...
proved.
```