SALENV

A "State Space Exploration Toolkit"
for SALENV version 1.0
July 2002

Leonardo de Moura

Acknowledgements

SALENV has been developed at SRI International in 2002.

I would like to express my gratitude to all members of the Computer Science Laboratory at SRI Internation. I also especially thank Natarajan Shankar and Harald Ruess for all useful remarks, help and suggestions. I also thank Manuel Serrano for his Scheme Compiler [Bigloo01] and Hans J. Boehm for his Garbage Collector [BoehmWeiser88, Boehm91].

This release of SALENV may still contain bugs. If you notice any, please forgive me and send a mail message to the following address: `demoura@csl.sri.com`.

New versions may be found at `http://www.csl.sri.com/sal`.

This is SALENV documentation version 1.0, July 2002.

1 Overview of SALENV

SALENV is a state space exploration toolkit which may be used to build model checkers, simulators, static debuggers, symbolic simulators, and other tools for the *SAL* specification language. Abstractly, the toolkit provides an API to traverse the concrete (or abstract) state space associated with a SAL specification. SALENV can also be seen as an *open model checker*, where a substrate of the implementation is open, allowing the user to extend, customize, and modify the implementation to affect *functionality* and/or *performance*. The toolkit is similar to an open compiler such as Open-C++ [OpenCpp], or an extensible application such as Emacs and AutoCAD. User extensions and modifications should be coded in the Scheme programming language. These extensions may be even compiled an linked with the kernel.

An explicit state model checker for the SAL language was built by using the SALENV toolkit. This model checker accepts correctness claims specified in the syntax of *Linear Temporal Logic* (LTL). The model checker can only handle models that are bounded, and have only countably many distinct behaviors. This mean that all correctness properties automatically become formally decidable, within the constraints that are set by the problem size and the computational resources that are available to the model checker to render the proofs. It is important to notice that the SALENV toolkit may be used to build different kinds of model checking tools, for instance, we are currently implementing a hybrid model checker, where part of the state is represented in a symbolic way.

2 Model-Checker

In this chapter we described the explicit state model checker for the SAL language which was built by using the SALENV toolkit.

2.1 Model-Checker Overview

This model checker accepts correctness claims specified in the syntax of *Linear Temporal Logic* (LTL). The model checker can only handle models that are bounded, and have only countably many distinct behaviors. However, we may use the model checker to detect errors in infinite state spaces. The output of the model checker is whether or not a given assertion holds. If the result is negative, a counter-example is produced.

We briefly highlight some of the features of this model-checker:

Extensibility
The user can modify/extend the behavior of the model checker.

Search Strategies
The user may select different strategies to traverse the specification state space. For instance, iterative depth first search may be used to find smaller counter examples.

Cache Representation
Different cache policies may be used. For instance, the supertrace algorithm used in the Spin model checker is basically a cache policy in our implementation.

State Representation
Different state representations may be used. Even a hybrid (explict + symbolic) state representation may be used.

Canonical term representation
A canonical representation of terms is available in the default execution engine. In this representation, term equality is equivalent to pointer equality. So, it is cheap to check when two terms/states are equal.

Dynamic scheduler
A SAL specification may contain code that can not be statically ordered. For instance,

```
X = IF A THEN NOT Y ELSE C ENDIF
Y = IF A THEN B ELSE X ENDIF
```

Here, there is no causal loop since `X` depends on `Y` only when `A` holds, and `Y` depends on `X` only when `NOT A` holds. This kind of code is mapped to a sequence of *actions* that are dynamically scheduled by the model checker, i.e. the execution order is only known at runtime.

2.2 A Brief Introduction to LTL

Linear temporal logic (LTL) was introduced to specify properties of the *traces* of a system. We define *trace* as an infinite sequence of states $(s_0, s_1, \ldots, s_i, \ldots)$, where s_{i+1} is a successor of s_i, i.e. there is a transition from s_i to s_{i+1}. If s_0 is a state, and $u = (s_1, s_2, \ldots, s_i, \ldots)$ is a trace, then $s_0.u = (s_0, s_1, s_2, \ldots, s_i, \ldots)$ is also a trace.

In our model checker, a LTL formula is composed of a finite set *Prop* which contains SAL expressions that describe atomic properties of states. For instance, the expression `pc1 = critical` may be an element of *Prop*. The standard boolean operators (`AND`, `OR`, `NOT`, `=>`, `<=>`) we can express static properties. For dynamic properties, we use the temporal operators: `X` (next), `U` (until), `W` (weak until), `R` (release), `F` (eventually), `G` (always), and `B` (before).

The semantics of LTL usually defines whether a trace `u` of a given system satisfies a formula. In our model checker, when you write:

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

The meaning is:

The theorem `mutualexclusion` is valid if and only if all traces `u` of `system` satisfy the LTL formula `G(NOT (pc1 = critical AND pc2 = critical))`.

The relation $u \models F$ (u satisfies F) is defined as:

- $u \models Expr$, if and only if $u = s_0.u'$ and *Expr* is true in s_0.
- $u \models NOT\ F$, if and only if it is *not* the case that $u \models F$.
- $u \models F\ OR\ F'$, if and only if $u \models F$ or $u \models F'$.
- $u \models X\ F$, if and only if $u = s_0.u'$ and $u' \models F$.
- $u \models U(F, F')$, if and only if:
  - $u \models F$, or
  - exists n such that $u = s_0.s_1 \ldots s_n.u'$, with $u' \models F'$ and for all $i \in \{0, \ldots, n\}$, $s_i \ldots s_n.u' \models F$.

The other operators may be seen as macros, and are defined in the following way:

```
TRUE = e OR (NOT e)
FALSE = NOT TRUE
e1 AND e2 = NOT ((NOT e1) OR (NOT e2))
R(e1, e2) = NOT U(NOT e1, NOT e2)
G(e) = R(FALSE, e)
F(e) = U(TRUE, e)
B(e1, e2) = R(e1, NOT e2)
W(e1, e2) = G(e1) OR U(e1, e2)
```

2.3 Useful LTL patterns

`G(Expr)`	`Expr` is always true.

`G(F(Expr))`
: `Expr` is true infinitely often.

`G(Expr1 => F(Expr2))`
: Everytime `Expr1` is true, eventually `Expr2` will also be true. This is a *response* formula.

`G(F(Cond-1)) AND G(F(Cond-2) ... AND G(F(Cond-n)) => Expr`
: `Cond-1`, . . . , `Cond-n` are *fairness* conditions. We say that `Expr` is true assuming these fairness conditions.

2.4 Counter-examples

A counter-example contains information about the transitions that were fired. These transitions can be specified by the following grammar:

```
transition ::= simple-transition
             | named-transition
             | multi-command-transition
             | asynch-transition
             | synch-transition

simple-transition ::= [ <context-name> : <line-number> ]
named-transition ::= <name> : <transition>
multi-command-transition ::= <transition> with [ <idx-values> ]
asynch-transition ::= < <transition> > . <num>
synch-transition ::= < <transitions> >

transitions ::= <transition>
              | <transition> <transitions>

idx-values ::= <idx-value>
             | <idx-value> , <idx-values>

idx-value ::= <idx-name> := <value>
```

In the `asynch-transition` rule, `<num>` specifies the module that executed the step. For instance, in `(mod1 || mod2)` `<num>=0` (`<num>=1`) tells us that `mod1`(`mod2`) performed a step.

Consider the module expression `(mod1 || (mod2 [] mod3)`, the transition `<starting:[ctx:48], <[ctx:102] with [x := 10]>.0>` should be interpreted in the following way:

- `mod1` executed the guarded command called `starting`, located in the context `ctx` at line 48.
- `mod2` executed the unnamed multi-command located in the context `ctx` at line 48, and the indexed variable `x` of the multi-command was assigned to 10.

The transition `<[ctx:54], <finishing:[ctx:301]>.1>` should be interpreted in the following way:

- `mod1` executed the unnamed guarded command located in the context `ctx` at line 54.
- `mod3` executed the guarded command called `finishing` in the context `ctx` at line 301.

2.5 SAL Model Checking: An Example

In this chapter we illustrate SAL model checking via a simplified version of Peterson's algorithm for 2-process mutual exclusion. 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*. A shared boolean variable, *turn*, arbitrates access to the critical section.

2.5.1 Developing the Specification

To begin, we will need to create a new SAL context. Therefore, let us create a file called '`tutorial1.sal`' which will contain the new context. Now we should define a new enumerated type called `PC`. This type consists of three values: `sleeping`, `trying`, and `critical`. After that, we should specify the two processes. Since the behavior of the two processes is quite similar, we can use a parametric SAL module (`mutex`) to specify them. In this way, `mutex[FALSE]` describes the behavior of the first process, and `mutex[TRUE]` the behavior of the other one. It is important to note that the variable `pc1` in the context `mutex` 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 counter-examples. So, we use the labels `init`, `enter_cs`, and `leave_cs`.

```
tutorial1 : CONTEXT =
BEGIN
  PC : TYPE = {trying, critical, sleeping};

  mutex [tval : BOOLEAN]: MODULE =
  BEGIN
    INPUT pc2 : PC
    OUTPUT pc1 : PC
    GLOBAL turn : BOOLEAN
    INITIALIZATION
      pc1 = sleeping;
      turn = FALSE
    TRANSITION
      [
        init: pc1 = sleeping --> pc1' = trying; turn' = tval
        []
        enter_cs: pc1 = trying AND (pc2 = sleeping OR turn /= tval) -->
          pc1' = critical
        []
        leave_cs: pc1 = critical --> pc1' = sleeping; turn' = tval
      ]
  END;
```

```
END
```

Initially, the program counter is set to `sleeping` and the variable `turn` to `FALSE`. 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 `mutex`.

```
system : MODULE =
  mutex[FALSE]
  []
  (RENAME pc2 TO pc1, pc1 TO pc2
   IN mutex[TRUE]);
```

2.5.2 Verifying the Specification

In this section, we will prove a mutual exclusion and a liveness property by using the sal model checker.

2.5.2.1 Mutual Exclusion

First, let us prove that is impossible for both processes to enter their critical section at the same time, which is precisely the objective in a mutual exclusion algorithm. This property may be stated in SAL in the following way:

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

Now, we can use the program `sal-model-checker` to check this property. `sal-model-checker` produces a program (*verifier*) to check a given property. The flag `--execute` will instruct the model checker to automatically execute the generated program after the generation phase.

```
sal-model-checker --execute tutorial1 mutualexclusion
⊣ verified
```

We may also check this property in two steps. In the first step, the model checker produces the *verifier* to check the given property. We use the flag `--output` to specify the name of the generated program.

```
sal-model-checker --output=checker tutorial1 mutualexclusion
```

Now, we can use the generated program to check the mutual exclusion property.

```
./checker
⊣ verified
```

The generated program contains several options, you can print them by using:

```
./checker --help
```

For instance, the option `--verbose` will print some statistics.

```
./checker --verbose
⊣ Checking...
⊣ verified
⊣ Number of visited states = 14
⊣ Maximum depth = 8
```

A more optimize program may be produce by using the option `--optimize`.

```
sal-model-checker --optimize --output=checker tutorial1 mutualexclusion
```

2.5.2.2 An Invalid Property

Now, let us try to prove an invalid property. This example will allow us to understand the counter-example generation in the SAL model checker. For instance, assume the following *invalid* property:

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

Now, we can generate the *verifier* for this property.

```
sal-model-checker --output=checker tutorial1 invalid
```

The following counter example is produced when we try to check the property. We may notice that there is an extra step in the counter-example. This extra step is considered because the an invariant may contain next variables such as: `pc1'`. Next variables contain the value of the given variable in the next state.

```
./checker
Counter example detected:
pc1 = sleeping
pc2 = sleeping
turn = FALSE
----------------
Transition: <init:[tutorial1:15]>.0
pc1 = trying
pc2 = sleeping
turn = FALSE
----------------
Transition: <enter_cs:[tutorial1:17]>.0
pc1 = critical
pc2 = sleeping
turn = FALSE
----------------
Transition: <init:[tutorial1:15]>.1
pc1 = critical
pc2 = trying
turn = TRUE
----------------
Transition: <leave_cs:[tutorial1:20]>.0
pc1 = sleeping
pc2 = trying
turn = FALSE
----------------
Transition: <init:[tutorial1:15]>.0
pc1 = trying
```

```
pc2 = trying
turn = FALSE
----------------
Transition: <enter_cs:[tutorial1:17]>.1
pc1 = trying
pc2 = critical
turn = FALSE
----------------
Transition: <leave_cs:[tutorial1:20]>.1
pc1 = trying
pc2 = sleeping
turn = TRUE
----------------
```

A smaller counter example may be produced by using an iterative depth first search. The flag `--step=<num>` activates this search strategy, and `<num>` specify the search depth increment in each iteration. In other words, the model checker will search until depth `<num>`, `2*<num>`, `3*<num>`, and so on.

```
./checker --step=1
```

```
Counter example detected:
pc1 = sleeping
pc2 = sleeping
turn = FALSE
----------------
Transition: <init:[tutorial1:15]>.1
pc1 = sleeping
pc2 = trying
turn = TRUE
----------------
Transition: <init:[tutorial1:15]>.0
pc1 = trying
pc2 = trying
turn = FALSE
----------------
Transition: <enter_cs:[tutorial1:17]>.1
pc1 = trying
pc2 = critical
turn = FALSE
----------------
Transition: <leave_cs:[tutorial1:20]>.1
pc1 = trying
pc2 = sleeping
turn = TRUE
----------------
```

We can use the option `--compact-trace` to request a more compact counter-example, where only the differences between two successors are printed.

```
./checker --step=1 --compact-trace
```

```
Counter example detected:
pc1 = sleeping
```

```
pc2 = sleeping
turn = FALSE
----------------
Transition: <init:[tutorial1:15]>.1
pc2 = trying
turn = TRUE
----------------
Transition: <init:[tutorial1:15]>.0
pc1 = trying
turn = FALSE
----------------
Transition: <enter_cs:[tutorial1:17]>.1
pc2 = critical
----------------
Transition: <leave_cs:[tutorial1:20]>.1
pc1 = trying
pc2 = sleeping
turn = TRUE
----------------
```

2.5.2.3 Liveness

It is important to note that there is several trivial algorithms that satisfy the mutual exclusion property. For instance, an algorithm that only allows one process to execute. Therefor, it is important to prove liveness properties. In our example, we can try to prove that a process is in the critical section infinitely often. We can state this property in the following way:

```
liveness1 : THEOREM system |- G(F(pc2 = critical));
```

Now, we generate the *verifier*.

```
sal-model-checker --output=checker tutorial1 liveness1
```

However, the following counter example is produced. In this counter-example, it is clear that only one process is executing. The problem is that SAL does not have a builtin fairness condition such as other model checkers. This decision allows the user to experiment different fairness conditions.

```
checker --step=1 --compact-trace
```

```
Counter example detected:
pc1 = sleeping
pc2 = sleeping
turn = FALSE
----------------
Transition: <init:[tutorial1:15]>.0
pc1 = trying
----------------
Transition: <enter_cs:[tutorial1:17]>.0
pc1 = critical
----------------
```

```
Transition: <leave_cs:[tutorial1:20]>.0
pc1 = sleeping
----------------
Transition: <init:[tutorial1:15]>.0
pc1 = trying
----------------
Transition: <enter_cs:[tutorial1:17]>.0
pc1 = critical
pc2 = sleeping
turn = FALSE
----------------
```

It is important to note that a nested depth first search algorithm is used to check more complicated LTL formulas. This algorithm contains two depth first searches, so it is not always possible to produce the smaller counter examples by using a nested iterative depth first search. However, usually a smaller `<num>` in `--step=<num>` will produce a smaller counter example. In a future version we will provide new search strategies that will not have this problem.

To check the intended liveness property, we should specify our own fairness condition. For instance, we can prove the following property:

```
liveness2 : THEOREM system |- G(pc2 = trying => F(pc2 = critical));
```

```
sal-model-checker --execute tutorial1 liveness1
 ⊣ verified
```

It is also possible to prove the following properties:

```
liveness3 : THEOREM system |- G(F(turn)) => G(F(pc2 = critical));
liveness4 : THEOREM system |- G(F(NOT turn)) => G(F(pc1 = critical));
```

2.5.3 Complete Description of the Specification

```
tutorial1 : CONTEXT =
BEGIN
  PC : TYPE = {trying, critical, sleeping};

  mutex [tval : BOOLEAN]: MODULE =
  BEGIN
    INPUT pc2 : PC
    OUTPUT pc1 : PC
    GLOBAL turn : BOOLEAN
    INITIALIZATION
      pc1 = sleeping;
      turn = FALSE
    TRANSITION
      [
        pc1 = sleeping --> pc1' = trying; turn' = tval
        []
        pc1 = trying AND (pc2 = sleeping OR turn /= tval) -->
```

```
            pc1' = critical
          []
          pc1 = critical --> pc1' = sleeping; turn' = tval
        ]
    END;

    system : MODULE =
      mutex[FALSE]
      []
      (RENAME pc2 TO pc1, pc1 TO pc2
       IN mutex[TRUE]);

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

    invalid : THEOREM system |- G(NOT (pc1 = trying AND pc2 = critical));

    liveness1 : THEOREM system |- G(F(pc2 = critical));

    liveness2 : THEOREM system |- G(pc2 = trying => F(pc2 = critical));

    liveness3 : THEOREM system |- G(F(turn)) => G(F(pc2 = critical));

    liveness4 : THEOREM system |- G(F(NOT turn)) => G(F(pc1 = critical));

  END
```

2.6 Available Options

The model checker has several option, you can type `sal-model-checker --help` to print them.

2.6.1 Compilation Time Options

`--version`
: Show the current sal-model-checker release.

`--verbose`
: Be verbose and print some statistics.

`--optimize`
: Apply some code transformations, and use all compiler optimization flags.

`--interpret`
: Interpret the generated code. This option is useful if you are still developing a specification, since the compilation of the generated code may be time consuming.

`--use-gmp`
: Enable the GNU multi-precision library. This option should be used if you are using real numbers or huge natural numbers. This feature, of course, implies performances penalty.

`--skip-transitions`
: Activate the SAL skip transitions, i.e. if it is impossible to perform a transition, then the model checker assumes a "silent" skip transition.

`--execute`
: Automatically execute the generated code.

`--output=<file>`
: Specify the name of the generated executable file. The default name is '`a.out`'. This option, of course, is only considered if you are compiling the generated code. The generated executable file also contains several options. Type `<file> --help` to print them.

`--disable-runtime-scheduler`
: Most of the specifications do not contain causal cycles, therefore, we may use this flag to disable the runtime scheduler. This option generates more efficient code. If your specification contains a causal cycle, and you use this option you will receive the "unable to execute" message.

`--user-check-state-fcn=<name>`
: User defined funtion name that will be called to check whether a state should be visited or not. The function must be defined in the file associated with the option `--user-code-file-name`.

`--user-code-file-name=<name>`
: User defined file name that will be inserted into the generated model checker.

`--user-library=<name>`
: User defined external library that should be linked with the generated model checker.

2.6.2 Execution Time Options

The following options are only considered if you are using the `--execute` flag. They are also available in the generated executable file.

`--dfs`
: Perform a depth first search (default).

`--bfs`
: Perform a bread first search. This option can only be used to check invariants.

`--idfs`
: Perform a iterative depth first search. This option is useful to produce smaller counter examples.

`--step=<num>`
: Step size in the iterative depth first search (default = 100).

`--max-depth=<num>`
: Maximum search depth. This option is useful if the state space is too huge.

`--cache-size=<size>`
: The initial cache size.

`--supertrace`
: Use the supertrace algorithm to perform the validation. Supertrace is a controlled partial-search technique that is only meant for the verification of specifications that cannot be analyzed exhaustively.

`--detailed-trace`
: Show the value of local variables when printing a counter example.

`--compact-trace`
: Show a compact counter example, only the differences between two successors are described.

`--tick=<num>`
: Tick frequency (default = 10000).

3 Emacs Front End

The command `salenv-fron-end` starts a customized version of the Emacs editor. This customization will add a menu and will automatically start `salenv`. Some keyboard commands are also defined.

3.1 Menu commands

help-salenv
: Opens the `info` version of this document.

parse
: Parses the context in the current buffer.

import
: Imports the SAL context in the current buffer. After executing this command, the abstract syntax tree of the given context will be available in the `salenv` process shell. The context will be parsed if necessary. This command will check common errors such as: undefined variables, missing contexts, etc.

generate verifier
: Generates a *verifier* program to check the property located in the current cursor position. If the cursor is not located inside a SAL assertion, then an error message will be produced. The name of the generated program is `checker`. The user may add extra options in the mini buffer.

execute verifier
: Executes the program `checker`. The user may add extra options in the mini buffer.

generate & execute
: Combines the two previous commands.

quick help
: Opens a *Man* page describing the model checker available options.

view state space (depth 5)
: Produces a graph representing the state space of the module located in the current cursor position. This command will search until depth = 5. This command is not intended to be used in large state spaces. For large state spaces, you should build your own specialized viewer. For instance ‘`pcp_viewer.scm`’ contains the code used to build a viewer for the priority ceiling protocol.

view state space (depth 10)
: Similar to the command above.

view state space (depth 15)
: Similar to the command above.

view state space
: Similar to the command above, but you should specify the maximum depth in the mini buffer.

reset Reinitializes the `salenv` process. This command is useful if the `salenv` process stopped to respond or generated a segmentation violation.

go to salenv shell
Goes to the buffer which is executing the `salenv` process.

go to shell Goes to the shell buffer and changes the current directory.

report-salenv-bug
Sets up mail buffer for reporting SALENV bugs.

exit-salenv
Closes the Emacs front end.

3.2 Keyboard commands

`C-c C-v` Generate and execute a verifier to check the property located in the current cursor position. If the cursor is not located inside a SAL assertion, then an error message will be produced. The name of the generated program is `checker`. The user may add extra options in the mini buffer.

`C-c C-e` Executes the program `checker`. The user may add extra options in the mini buffer.

`C-c C-g` Generates a *verifier* program to check the property located in the current cursor position. The user may add extra options in the mini buffer.

`C-c C-s` Produces a graph representing the state space of the module located in the current cursor position. This command is not intended to be used in large state spaces. For large state spaces, you should build your own specialized viewer. For instance ‘`pcp_viewer.scm`’ contains the code used to build a viewer for the priority ceiling protocol. The user should specify the maximum search depth in the mini buffer.

`C-c i` Opens the `info` version of this document.

`C-c C-z` Goes to the buffer which is executing the `salenv` process.

4 FAQ

4.1 Status of SALENV

SALENV originally was an acronym for SAL Environment. However, today, SALENV is a state space exploration toolkit for the SAL specification language. It was developed at SRI International's Computer Science Laboratory. SALENV 1.0 is the current version as of this writing. The program is still under development. So, you should expect bug and problems. If you notice any, please send a mail message to the following address: `demoura@csl.sri.com`.

4.2 General questions

4.2.1 How do I set up a '`.salrc`' file properly?

See Info file '`salrc`', node '`Init File`'

The SALENV customization file is '`.salrc`'. It should be located in your home directory. This file will be loaded when you start SALENV. The customization file should contain Scheme code, so consider taking a bit of time to learn Scheme. You can use all SALENV commands described in this manual.

4.2.2 How do I set up a '`.emacs`' file properly?

See Info file '`emacs`', node '`Emacs Init File`'

If you plan to run SALENV inside Emacs, you should also consider to make the following modifications in your '`.emacs`' file:

```
(add-to-list 'load-path "/usr/local/salenv/emacs")
(require 'sal-mode)
(setq auto-mode-alist
      (append
       '(("\\.sal$" . sal-mode))
         auto-mode-alist))
```

Note the code above assume that you installed SALENV in the */usr/local* directory.

4.2.3 How do I set up the `SALPATH`?

SALENV and related tools will search context files in the path *SALPATH*. The easiest way to set up this path is to set the environment variable *SALPATH*. If you are using *bash*, then you should modify your '`.bashrc`' file. For instance, to set the current and the */usr/local/salenv/lib* directories, you should add the following command to your '`.bashrc`' file:

```
export SALPATH=.:/usr/local/salenv/lib
```

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

```
setenv SALPATH  .:/usr/local/salenv/lib
```

You can also add new directories to the SALPATH when you are starting or running SALENV. You can add directory */XXX/YYY* to the load path like this:

```
(sal/add-to-path! "/XXX/YYY")
```

4.2.4 How do I start SALENV?

If you are using Emacs, then type *`M-x salenv`*.

4.2.5 How do I exit from SALENV?

You can type:

```
(exit)
```

or

```
(quit)
```

4.2.6 How can I get a more detailed error message?

You should use the Bigloo error notifier. To use it, you should add the following line to your '`.salrc`' file:

```
(set-repl-error-notifier! default-repl-error-notifier)
```

After that, you can set the environment variable *BIGLOOSTACKDEPTH*. So, an execution stack of depth *BIGLOOSTACKDEPTH* is printed when an error is raised. If you are using *bash*, then you can add the following command to your '`.bashrc`' file:

```
export BIGLOOSTACKDEPTH=12
```

4.2.7 How can I repeat the last command?

If you are running SALENV inside Emacs, you can type *`M-p`* and *`M-n`* to traverse the list of previously typed commands.

4.3 Compiling and Installing

4.3.1 What are the required third-party packages?

Required packages:

- GNU make (`gmake`).
- GNU C compiler (`gcc`).
- GNU Multi Precision library (`gmp`). `http://www.swox.com/gmp/`
- Bigloo Scheme Compiler. `http://www-sop.inria.fr/mimosa/fp/Bigloo/`
- A SAL parser (`sal2xml`). `http://www.csl.sri.com/sal/`

Optional packages:

- Graphviz (`dot`). `http://www.research.att.com/sw/tools/graphviz/download.html`

The SALENV `configure` script will check if you have these packages.

4.3.2 How do I compile and install SALENV and related tools?

This answer is meant for users of Linux systems. Users of other Unix-like systems may also try it.

For Linux systems, the easiest way is often to compile it from scratch. You will need:

- SALENV sources. Sources are available at `http://www.csl.sri.com/sal`.
- `gzip`, the GNU compression utility. You can get `gzip` via anonymous ftp at mirrors of '`ftp.gnu.org`' sites; it should compile and install without much trouble on most systems. Once you have retrieved the SALENV sources, you will probably be able to uncompress them with the command

  ```
  gunzip --verbose salenv-1.0.tar.gz
  ```

- '`tar`', the "tape archiving" program, which moves multiple files into and out of archive files, or "tarfiles." All of the files comprising the SALENV source come in a single tarfile, and must be extracted using '`tar`' before you can build SALENV. Typically, the extraction command would look like

  ```
  tar -xvvf salenv-1.0.tar
  ```

  The '`x`' indicates that we want to extract files from this tarfile, the two '`v`'s force verbose output, and the '`f`' tells '`tar`' to use a disk file, rather than one on tape.

  If you're using GNU '`tar`' (available at mirrors of '`ftp.gnu.org`'), you can combine this step and the previous one by using the command

  ```
  tar -zxvvf salenv-1.0.tar.gz
  ```

  The additional '`z`' at the beginning of the options list tells GNU tar to uncompress the file with gunzip before extracting the tarfile's components.

At this point, the SALENV sources should be sitting in a directory called '`salenv-1.0`'. On most common Linux and Unix-like systems, you should be able to compile SALENV with the following commands:

```
cd salenv-1.0       # change directory to salenv-1.0
./configure         # configure SALENV for your particular system
make                # use Makefile to build components, then SALENV
```

If the `make` completes successfully, the odds are fairly good that the build has gone well.

By default, SALENV is installed in the following directories:

'`/usr/local/bin`'
: binaries

'`/usr/local/salenv/`'
: Support files

'`/usr/local/info`'
: Info documentation

'`/usr/local/man`'
: Man documentation

'`/usr/local/lib/bigloo/2.5a`'
: Library files, if you installed Bigloo 2.5a at '`/usr/local`'.

To install files in those default directories, become the superuser and type

```
make install
```

Note that `make install` will overwrite '`/usr/local/bin/salenv`', SALENV Info/Man files, and all related libraries and executable files.

Much more verbose instructions (with many more hints and suggestions) come with the SALENV sources, in the file '`INSTALL`'.

4.3.3 How do I change the installation directory?

You can specify an installation prefix other than '`/usr/local`' by giving `configure` the option `--prefix=PATH`.

```
./configure --prefix=PATH
```

4.4 Bugs and problems

4.4.1 Error in Loading Shared Libraries

On some architecture you will be needing to tell the loader where to find the *SALENV* and *Bigloo* shared libraries. This can be done two ways:

- setting the shell `LD_LIBRARY_PATH` variable.
- updating the '`/etc/ld.so.conf`' file (read by ldconfig man page).

Otherwise, you will probably receive the following error message:

```
salenv: error in loading shared libraries: libsalcore.so: cannot open shared object
```

SALENV libraries are installed in the *Bigloo* shared libraries directory. If you installed Bigloo in '`usr/local`', then the libraries are located at:

```
/usr/local/lib/bigloo/2.5a
```

In this case, you can fix the problem by adding the following commando to your '`.bashrc`' file:

```
export LD_LIBRARY_PATH=/usr/local/lib/bigloo/2.5a:$LD_LIBRARY_PATH
```

4.4.2 Segmentation Violation and GMP

In some examples, if you use the flag `--use-gmp` in the SAL model checker, you will generate an executable file that will produce a segmentation violation. This problem is related to the GMP interface and the weak pointer module.

4.4.3 Segmentation Violtaion and the interpreter

salenv is a command language interpreter, and it is not intended to be used to check huge state spaces. If you try to model check huge state spaces using *salenv*, then you will probably produce a segmentation violation due to stack overflow.

4.4.4 Needed to allocate blacklisted block at 0x??????

The message "Needed to allocate blacklisted block at 0x???????" is produced by the Scheme garbage collector, and can be safely ignored. This message will be suppressed in future versions.

4.4.5 Value was not completely initialized

The *salenv* forces the user to initialize all local, global and output variables. If the user forgets to initialize these variables, the tool will produce the message "Value was not completely initialized".

4.4.6 Deadlocks in SAL specifications

A trace is in a deadlock if all guarded transitions are disabled. You can avoid deadlocks by using the option `--skip-transitions`, this option will add a skip transition when all guarded commands are disabled. To prove a LTL property, the SAL model checker generates a Buchi automata. This automata accepts infinite traces that are counter-examples for the LTL property. So, if all traces are in deadlock, the property is automatically valid, since there isn't any infinite trace. The `sal-deadlock-checker` can be used to detect deadlocks in a SAL specification.

4.5 Major packages and programs

4.5.1 salenv

salenv is a command language interpreter that executes commands read from the standard input. You may use *salenv* to test new scripts and execute any command described in the manual.

4.5.2 sal-model-checker

sal-model-checker is explicit state model checker for the SAL language. It was built by using the *SALENV* toolkit. This model checker accepts correctness claims specified in the syntax of *Linear Temporal Logic* (LTL). The model checker can only handle models that are bounded, and have only countably many distinct behaviors. This mean that all correctness properties automatically become formally decidable, within the constraints that are set by the problem size and the computational resources that are available to the model checker to render the proofs. To verify that a specification satisfies some property, the model checker transforms the negation of the LTL formula into a Buchi Automata, builds the product of that automaton with the specification, and checks this product for emptiness. If the product is not empty, then a counter-example is produced.

4.5.3 extsal2sal

extsal2sal transforms an extended SAL context in a pure SAL context. An extended SAL context is similar to a pure SAL context, but contains new builtin constants. For instance, an extended SAL context may contain LTL operators (X, U, R, F, G, B, W). The new builtin operators are described in a specific SAL context. For instance, the LTL operators are described in the '`ltl.sal`' context. This context is useful if we want to type-check the pure SAL context produced by this tool.

4.5.4 sal-dependencies

sal-dependencies collects all dependencies of a given context. In other words, it returns all contexts used directly or indirectly by the given context. This command is useful when one wants to generate dependencies automatically inside a Makefile.

4.5.5 sal-pretty-printer

sal-pretty-printer generates a "human readable" version of the XML file associated with a SAL context. This XML file represents the abstract syntax tree of the SAL context. In the SAL framework, each context is stored in a different XML file. *sal-pretty-printer* searchs a context, i.e. the associated XML file, in the path specified by the environment variable `SALPATH`.

4.5.6 sal-simulator

sal-simulator is a script environment, based on the Scheme programming language, that allows the user to execute a SAL specification. Any command available in "sal-default-engine.bgl" can be used in this interactive shell. The user can use this script environment to build build graphical simulation tools.

4.5.7 sal-deadlock-detector

sal-deadlock-checker is a deadlock detector for the SAL language. This program is useful for detecting traces that lead to deadlock situations, that is, a trace where it is not possible to fire any guarded command. Obviously, a SAL specification can deadlock only if skip transitions are ignored. A *skip transition* in a module is performed when all guarded commands cannot be fired. If a deadlock is detected, a counter-example is produced.

4.5.8 libsalcore

The *libsalcore* library contains:

- XML parser
- SAL abstract syntax API
- SAL Symbol Table
- Support code such as: growable vectors, bit-arrays, GMP interface, ordered sets, object system, and weak pointers.

4.5.9 libsaldefaultengine

The *libsaldefaultengine* library contains the default SAL execution engine. This engine is used to execute code generated by the predefined code generators. Users may extend or modify this engine.

Index

Table of Contents

Short Contents