

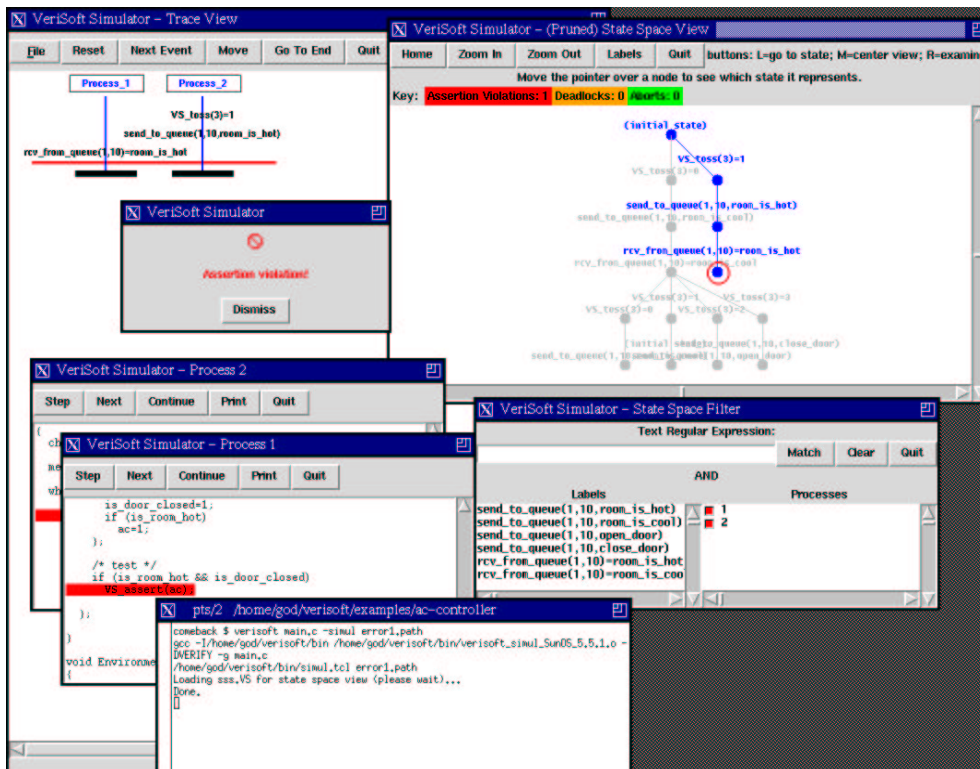
# VeriSoft Reference Manual

Patrice Godefroid  
Bell Laboratories, Lucent Technologies

February 19, 2001

## Abstract

This document describes VeriSoft version 2.x. More information on this project can be found at <http://www.bell-labs.com/projects/verisoft/>. Please direct your questions and comments to the author at [god@bell-labs.com](mailto:god@bell-labs.com).



# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Installation</b>	<b>5</b>
<b>3</b>	<b>Getting Started</b>	<b>5</b>
3.1	Design of an Air-Conditioning Controller . . . . .	5
3.2	Design of a Test Harness . . . . .	5
3.3	Running VeriSoft in Manual Simulation Mode . . . . .	9
3.4	Running VeriSoft in Automatic Simulation Mode . . . . .	11
3.5	Running VeriSoft in Guided Simulation Mode . . . . .	13
<b>4</b>	<b>State-Space Exploration with VeriSoft</b>	<b>13</b>
4.1	Definition of State Space . . . . .	13
4.2	State-Space Exploration . . . . .	16
<b>5</b>	<b>Using VeriSoft</b>	<b>17</b>
5.1	Running VeriSoft . . . . .	18
5.2	A Word of Caution . . . . .	18
5.3	Using the VeriSoft Simulator . . . . .	19
5.3.1	The Trace View . . . . .	19
5.3.2	The Process View . . . . .	20
5.3.3	The (Pruned) State-Space View . . . . .	20
5.3.4	The State Space Filter . . . . .	21
5.4	Using VeriSoft in State-Space Exploration Mode . . . . .	22
5.4.1	Types of Errors Detected . . . . .	22
5.4.2	Search Parameters . . . . .	22
5.4.3	Structural Properties . . . . .	25
<b>6</b>	<b>Customization</b>	<b>25</b>
6.1	Existing VeriSoft Libraries . . . . .	27
6.2	Developing New VeriSoft Libraries . . . . .	29



# 1 Introduction

*Concurrent systems* are systems composed of elements that can operate concurrently and communicate with each other. Each component can be viewed as a *reactive* system, i.e., a system that continuously interacts with its environment. An example is a telephone switching system, which performs many tasks concurrently and must interact with subscribers, technicians, and other switches. The software that controls a concurrent reactive system is notably hard to design and test because its components may interact in many unexpected ways. Traditional software testing techniques are of limited help since test coverage is bound to be only a minute fraction of the possible behaviors of the system. Furthermore, scenarios leading to errors are often extremely difficult to reproduce.

*VeriSoft* is a tool for systematically exploring the state spaces of systems composed of several concurrent processes executing arbitrary code written in any language. The *state space* of a system is a directed graph that represents the combined behavior of all the components of the system. Paths in this graph correspond to sequences of operations that can be observed during executions of the system. *VeriSoft* systematically explores the state space of a system by controlling and observing the execution of all the components, and by reinitializing their executions. It searches for coordination problems (deadlocks, divergences, etc.) between concurrent components, and for violations of user-specified assertions. *VeriSoft* can always guarantee a complete coverage of the state space up to some depth; hence, all possible executions of the system up to that depth are guaranteed to be covered.

Whenever an error is detected during state-space exploration, a scenario leading to the error state is saved in a file. Scenarios can be executed and replayed by the user with an interactive graphical simulator that can drive existing debuggers for examining the precise execution of the system processes.

*VeriSoft* also supports a special operation to simulate *nondeterminism*. This operation can be used to easily model the environment of the system being analyzed. For instance, a single nondeterministic test script can conveniently specify a family of thousands of test scripts that will be automatically generated, executed and evaluated by *VeriSoft*. This nondeterministic operation can be used anywhere in the system (any language, any procedure, any process).

*VeriSoft* has been applied successfully for analyzing (testing, debugging, reverse-engineering) several software products developed in Lucent Technologies, such as telephone-switching applications. Since *VeriSoft* can automatically generate, execute and evaluate thousands of tests per minute and has complete control over nondeterminism, it can *quickly* reveal behaviors that are virtually impossible to detect using conventional testing techniques.

## 2 Installation

Follow the instructions given in the file `INSTALL_README` of the distribution package.

## 3 Getting Started

Once VeriSoft has been installed, test your installation by performing the operations described in this section.

### 3.1 Design of an Air-Conditioning Controller

Imagine you have to design the software controlling an air-conditioning system. The controller reacts to external events in the form of messages sent by a thermometer and a (smart) door. A message is sent to the AC-controller when the room becomes hot or cool, or when the door is closed or opened. Your mission is to develop a piece of code that will turn on the air-conditioning system whenever the room is hot and the door is closed, and turn it off otherwise (to save energy).

Full of enthusiasm, you pick your favorite programming language, say C, and write the program shown in Figure 1.

Since you always plan ahead, you include at the end of the procedure an assertion which will test the correctness of your code.

### 3.2 Design of a Test Harness

In order to test your program, you decide to write a simple "test harness" that will run the above procedure in parallel with a little driver that will randomly generate messages to the controller. The code for the driver is depicted in Figure 2.

This code randomly picks a number between 0 and 3 using the special VeriSoft operation `"VS_toss(3)"` which simulates nondeterminism. At each iteration of the while loop, depending on the chosen number, the driver will send one of the four possible messages to the AC controller via a bounded FIFO queue.

In addition to the procedures `"AC_controller()"` and `"Environment()"`, the complete program you obtain also contains the code shown in Figure 3. The full program is in the file `main.c` in the directory `verisoft/examples/ac-controller`.

```

void AC_controller()
{
    char *message;

    int is_room_hot=0;    /* initially, room is not hot */
    int is_door_closed=1; /* and door is closed */
    int ac=0;            /* so, ac is off */

    while (1) {
        message=(char *)rcv_from_queue(to_me,QSZ);

        if (strcmp(message,"room_is_hot") == 0) {
            is_room_hot=1;
        };

        if (strcmp(message,"room_is_cool") == 0) {
            is_room_hot=0;
        };

        if (strcmp(message,"open_door") == 0) {
            is_door_closed=0;
            ac=0;
        };

        if ((strcmp(message,"close_door") == 0)){
            is_door_closed=1;
            if (is_room_hot)
                ac=1;
        };

        /* test */
        if (is_room_hot && is_door_closed)
            VS_assert(ac);
    };
}

```

Figure 1: C code for air-conditioning controller

```

void Environment()
{
    char *message;

    message=(char *)malloc(100);

    while (1) {

        switch(VS_toss(3)) {
        case 0: sprintf(message,"room_is_cool");
            break;
        case 1: sprintf(message,"room_is_hot");
            break;
        case 2: sprintf(message,"open_door");
            break;
        case 3: sprintf(message,"close_door");
            break;
        };

        send_to_queue(from_me, QSZ, message);
    };
}

```

Figure 2: C code for driver

```

/*
 * AC controller
 *
 */

#include <verisoft.h>
#include <stdio.h>
#include <string.h>

#define QSZ 10

int from_me, to_me; /* msqid */

void AC_controller();
void Environment();

void main()
{
    int i;

    from_me=get_bounded_queue(QSZ);
    to_me=get_bounded_queue(QSZ);

    if (fork()==0) {

        i=from_me; /* swap from_me and to_me */
        from_me=to_me;
        to_me=i;

        Environment();

    } else {

        AC_controller();

        wait(NULL);
        remove_queue(from_me);
        remove_queue(to_me);
        exit(0);
    };
}

```

Figure 3: Rest of the C code for obtaining a complete program

### 3.3 Running VeriSoft in Manual Simulation Mode

You are now ready to run the above program `main.c`. If you do not have write permission in the directory `verisoft/examples/ac-controller`, create a new directory and copy all the files from `verisoft/examples/ac-controller` to this new directory.

Since you have never run the program `main.c` before, the first thing you want to do is to make sure that the program behaves like you think it should behave for some simple scenarios. Therefore, you start by running VeriSoft on this program in manual simulation mode.

Type:

```
verisoft main.c -simul
```

This command compiles the program `main.c`, links it with the appropriate VeriSoft libraries, and then run VeriSoft in manual simulation mode. The windows shown in Figure 4 appear on your screen.

The “Trace View” on the left shows the current scenario, or “trace”, executed from the initial state of the system. This view only displays *visible* operations, according to the VeriSoft terminology. Visible operations include the operations “VS\_toss”, “VS\_assert” and operations on communication objects such as “send\_to\_queue” and “rcv\_from\_queue”. Initially, no visible operations are displayed since no such operations have been executed.

For each process in the system, a separate window is created. Each “Process View” shows the current state of the corresponding process. A process whose next instruction is colored in red is currently blocked.

Pressing the button “Step” or “Next” in a Process View window executes the next line of the process, stepping into or over function calls respectively. Pressing “Continue” resumes the execution of the process until the next visible operation is reached. After selecting a variable name with the mouse, pressing the button “Print” creates a new window where the value of the selected variable is printed.

The buttons “Reset”, “Next Event”, “Move” and “Go To End” of the Trace View can be used to drive the system in any of the intermediate states reached during the execution of the scenario displayed in the Trace View. The “File” menu enables you to save and load scenarios.

If you press the “Continue” button of Process 2, a little window, shown in Figure 5, pops up. This window prompts you to select one of the four possible choices for the nondeterministic operation “VS\_toss(3)” in Process 2. Indeed, in manual simulation mode, VeriSoft lets you drive the execution of the whole system yourself, “manually”.

Press a few times the “Continue” button of Process 1 or 2, and observe how VeriSoft reacts. The visible operations that have been executed are displayed in the Trace View.

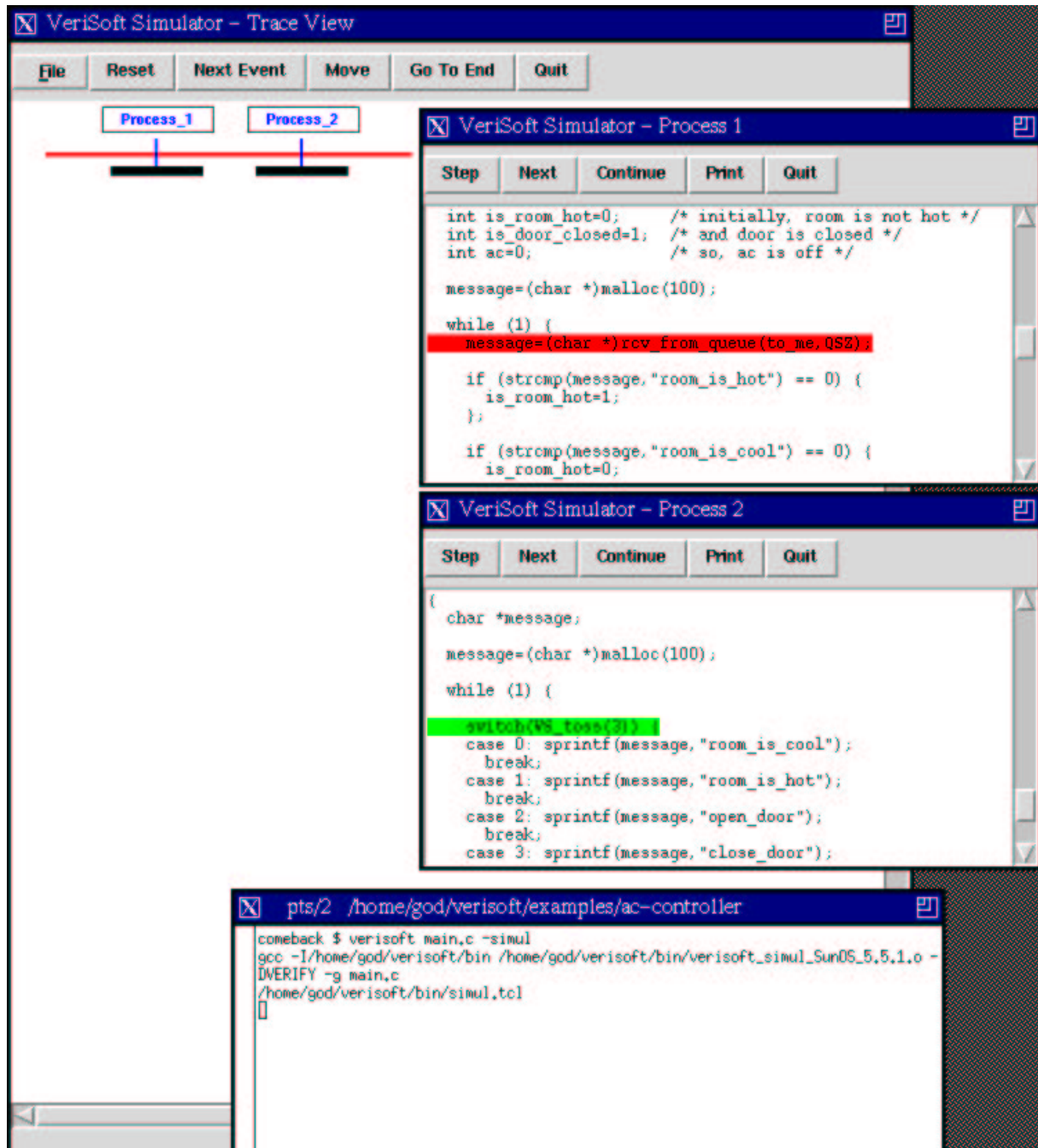


Figure 4: Screenshot of VeriSoft in manual simulation mode

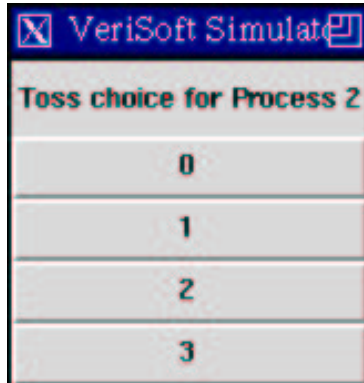


Figure 5: Toss choice

The red horizontal bar in the Trace View indicates the position of the current state in the scenario. Click on the “Move” button of the Trace View, move the cursor along the scenario displayed in the Trace View, and see how the red bar is moving along. Click on the left button of the mouse to select a position in the scenario. The whole system is then brought back to the state you have selected.

All the commands associated to buttons of the VeriSoft simulator are precisely described later, in Section 5.3.

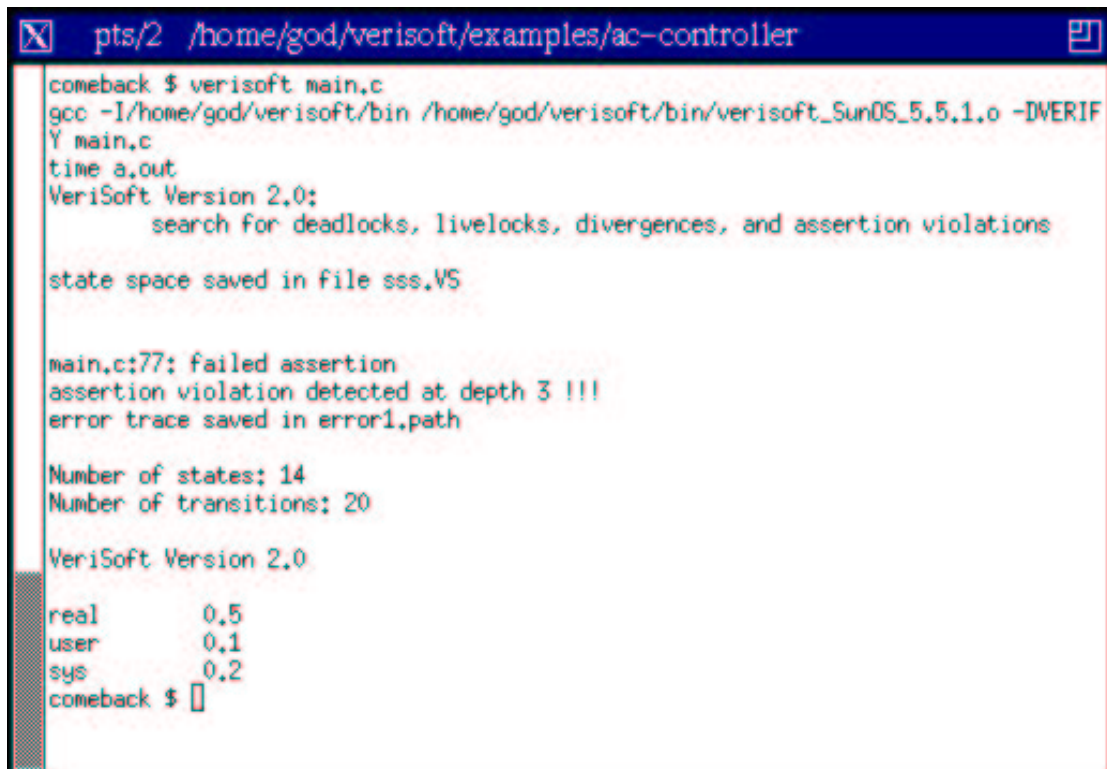
### 3.4 Running VeriSoft in Automatic Simulation Mode

It is time to challenge the correctness of your code using VeriSoft in automatic simulation mode.

In this mode, VeriSoft systematically explores all the possible executions of the system. The superposition of all these executions can be represented by a graph, called the “state space” of the system. Nodes in the graph correspond to states of the whole system, while edges correspond to transitions from states to states.

In practice, the state space of a software application is usually huge, and VeriSoft cannot explore it exhaustively. However, VeriSoft can easily explore hundreds of thousands scenarios in a reasonable amount of time (in one night, for instance), and examine millions of states and transitions completely automatically. VeriSoft can also always guarantee a complete coverage of the state space up to some depth, where the depth in the state space is measured by the number of visible operations executed from the initial global state. In other words, all the possible executions of the system up to that depth are guaranteed to be covered. See Section 4 for a precise definition of state space and for more information on the state-space exploration techniques used by VeriSoft.

Press the “Quit” button in any of the windows to quit the VeriSoft simulator. Then, run



```
pts/2 /home/god/verisoft/examples/ac-controller
comeback $ verisoft main.c
gcc -I/home/god/verisoft/bin /home/god/verisoft/bin/verisoft_SunOS_5.5.1.o -DVERIF
Y main.c
time a.out
VeriSoft Version 2.0:
    search for deadlocks, livelocks, divergences, and assertion violations

state space saved in file sss.VS

main.c:77: failed assertion
assertion violation detected at depth 3 !!!
error trace saved in error1.path

Number of states: 14
Number of transitions: 20

VeriSoft Version 2.0

real    0.5
user    0.1
sys     0.2
comeback $
```

Figure 6: Screenshot of VeriSoft in automatic simulation mode

VeriSoft in automatic simulation mode by typing:

```
verisoft main.c
```

After a few seconds, VeriSoft prints the diagnosis shown in Figure 6 and stops.

By default, VeriSoft systematically searches the state space of an application for deadlocks, livelocks, divergences, and assertion violations. (These types of errors are precisely described in Section 5.4.) It performs a sort of breadth-first search in the state space of the system: it systematically generates, executes and evaluates all the possible “short” scenarios that can be exhibited by the system, and progressively increases the length of the scenarios being tested.

In the case of our air-conditioning controller program, VeriSoft immediately found a scenario leading to an assertion violation. This “error trace” is saved in a file named “error1.path”.

Several options are available for searching for a specific type of errors, for controlling the search strategy, for starting a search from a specific initial state, etc. These options are discussed in Section 5.4.2.

### 3.5 Running VeriSoft in Guided Simulation Mode

Let us now examine the scenario generated by VeriSoft that led to an assertion violation. This scenario can be replayed using VeriSoft in guided simulation mode by typing:

```
verisoft main.c -simul error1.path
```

Several windows appear on your screen. The “(Pruned) State-Space View” shows the state space that has been explored in systematic state-space exploration mode (saved in the file `sss.VS`). After zooming in, centering the view, pressing the button “Labels” to view labels of transitions, and clicking with the left button on the state circled in red (i.e., the assertion violation), you obtain the view shown in Figure 7.

The “State Space Filter” window can be used to filter information displayed in the State-Space View, for instance by highlighting all the transitions executed by a specific process or corresponding to a specific visible operation.

Using all the linked views of the VeriSoft simulator, you can now replay (instruction per instruction if necessary) the scenario leading to the assertion violation, examine the state of the processes, navigate through the state space, explore alternative scenarios, etc., in order to discover the precise dynamic behavior of the application.

As an exercise, try to fix the code of the procedure `AC_controller()`. Then, run VeriSoft in automatic simulation mode to test whether your solution is correct. (A solution to this exercise is given in the file `verisoft/examples/ac-controller/main2.c`.)

**Note 1** In order to speed up the search, VeriSoft uses “smart” state-space search algorithms for dynamically pruning the state space in a completely reliable way: pruned parts of the state space are guaranteed not to contain any errors of the types discussed above. These parts are not displayed in the “(Pruned) State-Space View”. The search algorithms used in VeriSoft are discussed in Section 4.2. ■

## 4 State-Space Exploration with VeriSoft

In this section, we precisely define the key notions necessary to understand what VeriSoft does and how it works.

### 4.1 Definition of State Space

VeriSoft [God97] is a tool for systematically testing concurrent systems composed of a finite set of *processes* and a finite set of *communication objects*. Each process executes a sequence of

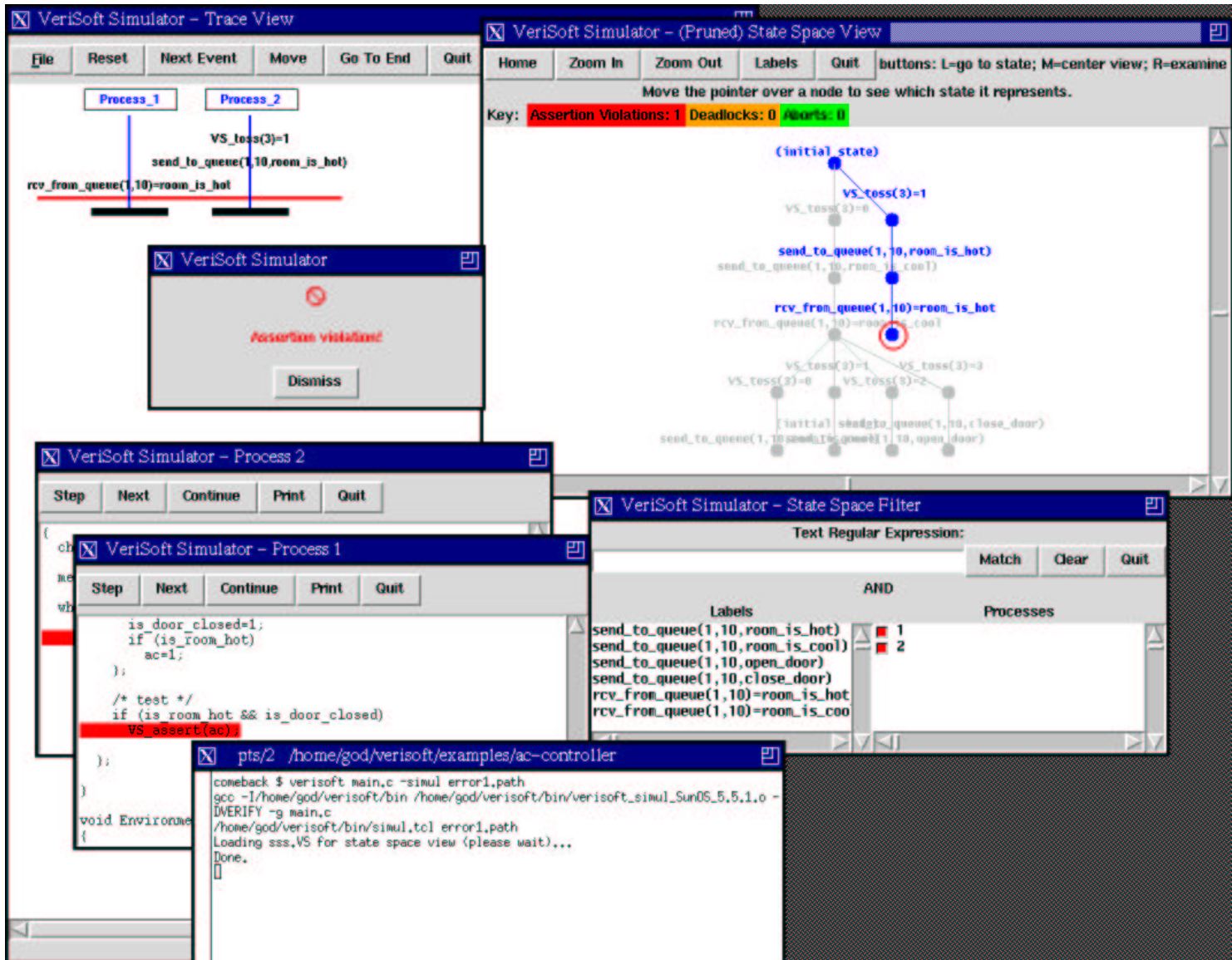


Figure 7: Screenshot of VeriSoft in guided simulation mode

*operations*, that is described in a sequential program written in a full-fledged programming language such as C or C++. Such sequential programs are assumed to be deterministic: every execution of the program on the same input data performs the same sequence of operations. We assume that processes communicate with each other by performing operations on communication objects, such as semaphores, and FIFO buffers, for example. We assume that operations on communication objects are executed atomically: namely, no process can execute an operation on a given communication object while another process is currently doing so. Operations on communication objects are called *visible operations*, while other operations are by default called *invisible*. The execution of an operation is said to be *blocking* if it cannot currently be completed; for example, waiting for the reception of a message blocks until a message is received. We assume that only executions of visible operations may be blocking.

The concurrent system is said to be in a *global state* when the next operation to be executed by every process in the system is a visible operation. Every process in the system is expected to always eventually attempt to execute a visible operation. When a process does not attempt to perform a visible operation within a given (user-specified) amount of time, VeriSoft reports an error called “divergence”. This implies that initially, after the creation of all the processes of the system, the system may reach a first and unique global state, called the *initial global state* of the system. We define a *transition* as a visible operation followed by a finite sequence of invisible operations performed by a single process and ending just before a visible operation. The *state space* of the concurrent system is composed of the global states that are reachable from the initial global state, and of the transitions that are possible between these.

A concurrent system as defined here is a closed system: from its initial global state, it can evolve and change its state by executing transitions. Thus, given a single “open” reactive system, the environment in which this system operates has to be represented, possibly using other processes, in order to close the system. For this purpose, a special operation “VS\_toss” is available to express a valuable feature of modeling languages, not found in programming languages: *nondeterminism*. This operation takes as argument a positive integer  $n$ , and returns an integer in  $[0, n]$ . The operation is visible and nondeterministic: the execution of a transition starting with VS\_toss( $n$ ) may yield up to  $n + 1$  different successor states, corresponding to different values returned by VS\_toss.

It can be shown [God97] that *deadlocks* and *assertion violations* can be detected by exploring only the global states of a concurrent system that satisfies the above assumption. Deadlocks are states where the execution of the next operation of every process in the system is blocking. Deadlocks are a notorious problem in concurrent systems, and are extremely difficult to detect through conventional testing methods. Assertions can be specified by the user with the special operation “VS\_assert”. This operation can be inserted in the code of any process, and is considered visible. It takes as its argument a boolean expression that

can test and compare the value of variables and data structures local to the process. When “VS\_assert(expression)” is executed, the expression is evaluated. If the expression evaluates to false, the assertion is said to be *violated*. Many undesirable system properties – such as unexpected message receptions, buffer overflows, and application-specific error conditions – can easily be expressed as assertion violations.

## 4.2 State-Space Exploration

VeriSoft is a tool for systematically exploring the state space of a concurrent system as defined above. In a nutshell, systematic state-space exploration is performed by controlling and observing the execution of all the visible operations of all the concurrent processes of the system. Every process of the concurrent system to be analyzed is mapped to a UNIX process. The execution of the system processes is controlled by an external process, called the *scheduler* (see Figure 8). This process observes the visible operations performed by processes inside the system, and can suspend their execution. By resuming the execution of (the next visible operation of) one selected system process in a global state, the scheduler can explore one transition between two global states in the state space of the concurrent system. By reinitializing the system, the scheduler can explore alternative paths in the state space.

Note that there are exactly two sources of nondeterminism in the concurrent systems we consider here, namely concurrency and VS\_toss operations. Since the VeriSoft scheduler has complete control over both, it is able to completely reproduce any scenario leading to an error found during a state-space search.

It is worth emphasizing that, obviously, the validity of the results reported by VeriSoft is guaranteed only if the concurrent system being tested satisfy all the assumptions presented in the previous section. For instance, if the system being analyzed contains components that behave nondeterministically and whose nondeterminism is not known to (controlled by) VeriSoft, VeriSoft might then miss errors it would detect otherwise, or might not be able to reproduce scenarios leading to erroneous states.

Since states of programs written in programming languages can be very complex (because of pointers, dynamic memory allocation, large data structures of various shapes, recursion, etc.), the VeriSoft scheduler does not attempt to compute any representation for the reachable states of the system being analyzed, and hence performs a systematic state-space exploration without storing any intermediate states in memory. It is shown in [God97] that the key to make this approach tractable is to use a new search algorithm built upon existing state-space pruning techniques known as partial-order methods [God96]. For finite acyclic state spaces, this search algorithm is guaranteed to terminate and can be used for detecting deadlocks and assertion violations without incurring the risk of any incompleteness in the verification results. In practice, VeriSoft can be used for systematically and efficiently testing the correctness of any concurrent system, whether or not its state space is acyclic. Indeed,

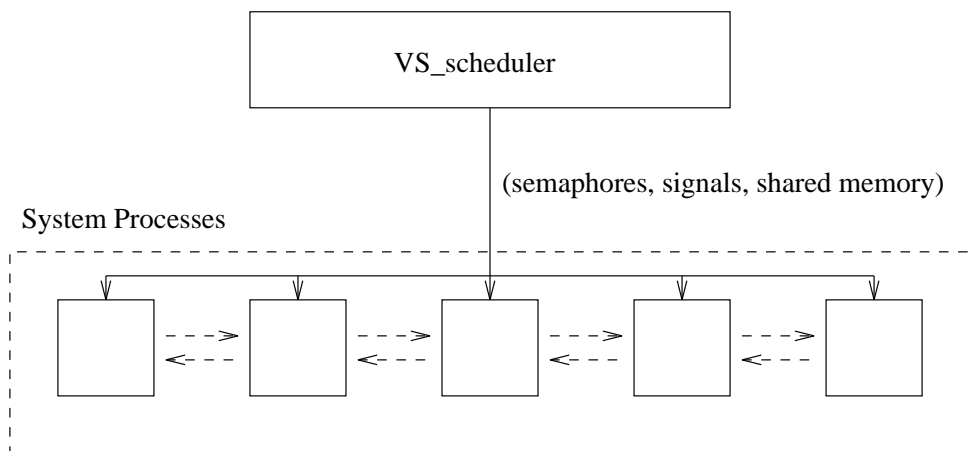


Figure 8: Overall Architecture of VeriSoft (State-Space Search Mode)

it can always guarantee, from a given initial state, complete coverage of the state space up to some depth. This is in contrast to conventional testing methods, in which only very few scenarios are explored.

When an error of the type listed above is detected during state-space exploration, a scenario leading to the error state is exhibited to the user. An interactive graphical simulator/debugger is also available for replaying scenarios and following their executions at the instruction or procedure/function level. Values of variables of each process can be examined interactively. Using the VeriSoft simulator, the user can also explore any path in the state space of the system with the same set of debugging tools.

## 5 Using VeriSoft

The distribution package available from the VeriSoft web-site contains a version of VeriSoft for analyzing multi-process C or C++ programs whose processes communicate via a predefined set of types of communication objects, namely semaphores and bounded queues, using the functions defined in the files `semdef.h` and `msgdef.h` of the `verisoft/bin` directory. This version of VeriSoft is for evaluation purposes and educational use only. Please contact the author for information concerning the availability of VeriSoft for other platforms (SGI, HP, AIX, ...), other languages (Java, Tcl, Perl, ...), and other libraries of communication objects (sockets, shared memory, files, timers, ...).

This section discusses how to use VeriSoft for analyzing the examples included in the distribution package.

## 5.1 Running VeriSoft

First, make sure that the full pathname of the “verisoft/bin” sub-directory is included in your environment variable PATH.

As mentioned in Section 3, VeriSoft can be run in 3 modes.

- `verisoft <program>.c -simul`  
runs VeriSoft in manual simulation mode.
- `verisoft <program>.c`  
runs VeriSoft in automatic simulation mode (systematic state-space search).
- `verisoft <program>.c -simul <error_file>.path`  
runs VeriSoft in guided simulation mode.

The script `verisoft` is an expectk script that compiles the C program to be analyzed (using `gcc`), links it with the appropriate VeriSoft libraries, and then executes VeriSoft in the appropriate mode.

You should always run VeriSoft from a directory you can write to, since the tool automatically saves scenarios leading to errors in the current directory. VeriSoft always expects to find in the current directory two files:

- `system_file.VS` is used to specify search parameters (see Section 5.4.2); and
- `a.out` is the main executable file for the application being tested.

## 5.2 A Word of Caution

A word of caution is necessary. VeriSoft and the examples provided in the distribution package use Unix interprocess communication resources such as Unix semaphores and message queues. Such resources are not “freed” automatically upon the termination of a program or even upon the termination of a login session – the application using these resources must free them explicitly before it terminates. If you have never used Unix interprocess communication facilities, you should familiarize yourself with this subject before starting developing your own concurrent C programs.

If you create communication objects using the functions `semget`, `get_bounded_queue` (defined in the file `msgdef.h` in `verisoft/bin`) and `shmget`, VeriSoft will keep track of the usage of these resources, and should always clean the operating system for you at the end of each run, including in the case of an “abnormal termination”, such as a segmentation fault. However, do not forget to clean the system yourself in the “production” version of your code, which will run without the supervision of VeriSoft.

The two main Unix commands for keeping track of the usage of these interprocess communication (IPC) resources and for cleaning the system manually are, respectively, `ipcs` and `ipcrm`. Consult the man pages on these commands for more information.

## 5.3 Using the VeriSoft Simulator

In manual and guided simulation modes, VeriSoft lets you control interactively the execution of all the processes of the system being analyzed. This is done through a graphical user interface which we call the VeriSoft simulator. This section describes the functions associated with the buttons of the windows of the VeriSoft simulator.

Let us consider successively the windows displayed in the screenshot of the VeriSoft simulator shown on the cover page of this document (same as Figure 7).

### 5.3.1 The Trace View

At any time, the “Trace View” of the veriSoft simulator shows the current “trace” (scenario) executed from the initial global state of the system. Only visible operations (cf. Section 4.2) are displayed. The red horizontal bar indicates the position of the current state in the scenario.

The following commands are available.

- **File Menu, Load:** Load a scenario from a “.path” file. Only scenarios saved in the “.path” format can be replayed by VeriSoft.
- **File Menu, Save:** Save the current (complete) scenario in a “.path” file, or a postscript file in black-and-white or color. An optional comment (e.g., title) can be inserted at the top of the scenario in postscript format.
- **Reset:** Reset the execution of the whole system back to its initial global state.
- **Next Event:** Continue the execution of the current scenario until the next visible operation is reached.
- **Move:** Move to any intermediate global state of the current scenario. Click on the “Move” button, then move the cursor in the scenario to the state you want to go to, and click on the left button of the mouse to select that state.
- **Go To End:** Go to the end of the current scenario.
- **Quit:** Quit VeriSoft.

### 5.3.2 The Process View

A “Process View” window is created for every process in the system. This view is used to display the current state of the corresponding process. The next line to be executed by the process is highlighted in the source code and colored using the following convention.

- **Green:** The next instruction is non-blocking.
- **Red:** The next instruction is blocking.
- **Blue:** Indicate the next instruction specified in the current scenario being played. This instruction is always non-blocking, by definition.
- **Yellow:** Indicate that the next instruction of the process cannot be executed by the VeriSoft simulator since another process is currently about to execute a non-visible operation. Indeed, the VeriSoft simulator does not let you explore simultaneously the internal behavior of more than one process at a time.

The following commands are available.

- **Step:** Execute the next line of the process, stepping into function calls.
- **Next:** Execute the next line of the process, stepping over function calls.
- **Continue:** Continue the execution of the process until the next visible operation is reached.
- **Print:** Print the value of a selected variable. Selecting a variable is done by dragging the cursor over the name of that variable in the program text while pressing on the left button of the mouse, or by double-clicking on the variable name with the left button of the mouse.
- **Quit:** Quit VeriSoft.

### 5.3.3 The (Pruned) State-Space View

The “(Pruned) State-Space View” shows the state space saved in the file `sss.VS` in the current directory. This file can be generated by running VeriSoft in systematic state-space exploration mode (see Section 5.4), and represent the part of the state space that has been explored. Assertions violations, deadlocks and abort states (see Section 5.4.1) are highlighted in red, orange and green, respectively. Note that the parts of the state space that have been pruned by the search algorithm used in VeriSoft (as explained in Section 4.2) are not displayed in the view.

The following commands are available.

- **Home:** Restore the initial view.

- **Zoom In:** Zoom in.
- **Zoom Out:** Zoom out.
- **Labels:** Show/hide the labels of the transitions. The label of a transition is the visible operation corresponding to that transition (see Section 4.2).
- **Quit:** Quit VeriSoft.

Additional commands can be executed by clicking on a state with the following mouse buttons.

- **Left Button:** Drive the whole system to the selected state.
- **Middle Button:** Center the view on the selected state.
- **Right Button:** “Examine” the selected state by centering the view on it and zooming in to a predefined level of detail.

#### 5.3.4 The State Space Filter

The “State-Space Filter” window highlights specific transitions in the State-Space View by painting these transitions in black. A transition is highlighted when both its label AND its process number are selected. Multiple labels and process numbers can be selected/unselected by clicking on the corresponding entry in the lists displayed in the State Space Filter window. Selected entries are painted in black.

Labels can also be selected via pattern matching by typing a regular expression in the space provided and then pressing the “Match” button: all the labels matching the regular expression will then be selected. Any regular expression accepted by the Unix command “regexp” can be specified. Type `man regexp` for a detailed description of valid regular expressions. Frequent regular expressions include “\*” (matches all labels), and “abc” (matches any label containing the string “abc”).

Pressing the “Clear” button unselects all previous selections. Pressing the “Quit” button quits VeriSoft.

Clicking on the square in front of a process number closes/opens the corresponding Process View window.

Remember to select labels AND process numbers for highlighting transitions in the State-Space View.

## 5.4 Using VeriSoft in State-Space Exploration Mode

### 5.4.1 Types of Errors Detected

In this mode, VeriSoft systematically searches the state space of a system for four types of errors.

- **Deadlocks:** A deadlock is a state where the next instruction of every process is blocking.
- **Livelocks:** A livelock occurs when a process has no enabled transition during a sequence of more than a given (user-specified) number of successive states in the state space.
- **Divergences:** A divergence occurs when a process does not attempt to execute any visible operation for more than a given (user-specified) amount of time.
- **Assertion violations:** An assertion specified using the visible operation “VS\_assert” is violated when its argument evaluates to false (zero).

Whenever an error is detected during state-space exploration, a scenario leading to the error state is saved in a “.path” file. Such scenarios can be replayed using the VeriSoft simulator. At the end of the search, VeriSoft prints the number of states and transitions that have been explored.

### 5.4.2 Search Parameters

Several parameters for controlling a state-space search can be specified in the file `system_file.VS` that VeriSoft always expects to find in the current directory. A sample `system_file.VS` file is included in the `verisoft` directory of the distribution package, and is shown in Figure 9 and 10.

The parameters that can be defined in the file `system_file.VS` are the following.

- `nproc` is the number of processes in the system. `nproc` is the only parameter that *must* be defined in `system_file.VS`. This number specifies the number of processes visible to VeriSoft when the system reaches its initial global state. See Section 6 for more information on how to set this parameter for your own programs.
- `max_depth` is the maximum depth of the states being searched in the state space. Remember the depth in the state space is measured by the number of visible operations executed from the initial global state of the system. States that are reachable only by scenarios longer than `max_depth` will not be explored.

```

#
# The value of parameters for VeriSoft can be specified in this file,
# as a list of pairs 'parameter_name value'.
#
# Warning: maximum one such pair per line!
#
# The pattern '#' starts a comment -- the rest of the line is ignored.
#

nproc 5 # MANDATORY! nbr of processes in system (must be exact!)

max_depth 100 # max depth of search (default 100)

depth_incr 5 # depth of one layer of breadth-first search (default 5)

ignore_deadlock 0 # if 1, deadlocks are not reported (default 0)

livelock_limit 15 # max number of successive trans during which a process
                  # might be blocked without having a livelock (default 15)

divergence_limit 10 # max delay (in sec) before divergence (default 10; minimum is 10)

# init_path filename.path # scenario to define initial state (default none)

stop_at_error 1 # nbr of errors (excluding divergences) before search stops (default 1)

save_state_space 0 # if 1, state space is saved in file sss.VS (default 0, recommended 1)

max_sss 10 # max size of sss.VS in megabytes (default 10)

# stop_at_state 10000 # max nbr of states explored during search (default no limit)

# random_seed 40 # seed (any int) to turn on random search mode (default 1)

# sigusr2 1 # if 1, SIGUSR2 instead of SIGKILL is sent to terminate system processes
            # (SIGUSR2 can then be trapped and trigger cleanup code) (default 0)

```

Figure 9: Part 1 of sample file system\_file.VS

- `depth_incr` is the depth of the layers of the breadth-first search performed by VeriSoft. For instance, if `depth_incr` is set to 5, VeriSoft will start by exploring all the possible paths of length  $\leq 5$  in the state space, then will explore the paths up to length  $\leq 10$ , then  $\leq 15$ , and so on until `max_depth` is reached. Note that if `depth_incr` is equal to `max_depth`, VeriSoft will perform a depth-first search until depth `max_depth`.

**Tip:** if you believe that error states are likely to be located beyond a specific depth in the state space (for instance, if some initialization procedure has to be completed before there is a chance of hitting an assertion violation), it is a good idea to set `depth_incr` to a value greater than this depth in order to increase the chances of finding bugs quickly.

- `ignore_deadlock` determines whether or not deadlocks should be ignored. By default, `ignore_deadlock` is 0, which means that deadlocks are reported.
- `livelock_limit` is the maximum number of successive transitions during which a process might be blocked without having a livelock. For instance, if `livelock_limit` is 15, a livelock will be reported by VeriSoft if there is a path in the state space during which a process is continually blocked during at least 15 consecutive transitions.
- `divergence_limit` is the maximum delay (in seconds) given to any process to reach its next visible operation. When this delay expires, a divergence is reported by VeriSoft.
- `init_path` defines a scenario specified in a “.path” file for driving the system to a global state from which the state-space search starts. This parameter can be used to start a state-space search from any specific global state. This is useful for focusing a search on a specific area of the state space.
- `stop_at_error` is the number of errors that are reported before the state-space search stops. By default, `stop_at_error` is 1, which means that VeriSoft stops its search as soon as a first error is detected.

Note that VeriSoft never continues a search after it has detected a divergence. Indeed, a divergence may be due to VeriSoft losing the control of a process of the system being analyzed. If this process still exists and use interprocess-communication resources, VeriSoft cannot guarantee anymore the proper control of the system processes. In that case, you may have to “clean” the operating system manually, using the Unix command `kill` and `ipcrm` (see Section 5.1).

- `save_state_space` determines whether or not the state space should be saved in the file `sss.VS` in the current directory. Since state spaces can be very large, `save_state_space` is set to 0 by default for “safety” reasons, which means that the state space is not saved in `sss.VS`. However, if you have several megabytes of disk space available, we do recommend you to set `save_state_space` to 1 in order to increase the feedback you get from VeriSoft. Indeed, especially when VeriSoft did not find any errors, examining the state space can reveal something wrong in the way the state-space search was set

up, or provide useful coverage information about the visible operations that have been exercised during the search.

- `max_sss` is the maximum size (in megabytes) that the file `sss.VS` can reach.
- `stop_at_state` is the maximum number of states that can be explored during a state-space search. By default, there is no limit in the number of explored states. This parameter can be used to force the automatic termination of a state-space search, which is useful when doing regression testing for instance.
- `random_seed` is a seed number used to randomize the order in which transitions due to different `VS_toss` return values are explored. When the state-space of a system using `VS_toss` is so large that it cannot be completely explored, this option is useful to force multiple runs to visit different subsets of the state space.
- `sigusr2` determines whether a `SIGKILL` or a `SIGUSR2` signal should be sent by VeriSoft to terminate each system process. By default, `sigusr2` is 0, and a `SIGKILL` signal is sent. If `sigusr2` is 1, a `SIGUSR2` signal is sent to each system process when VeriSoft needs to terminate their execution; it is then the responsibility of each of these processes to intercept the `SIGUSR2` signal, execute some clean-up code if necessary, and then terminate.

### 5.4.3 Structural Properties

Properties of the structure of the system being analyzed can also be specified in the file `system_file.VS`. These properties are exploited by the search algorithms used in VeriSoft for further pruning the state space. If any of such declared “structural properties” is detected to be violated during the search, an error is reported. Whenever possible, we strongly recommend to declare as many structural properties as possible. Indeed, such properties can sometimes contribute to speed up the search by several orders of magnitude.

Structural properties are all of the form

“operation  $x$  is allowed to be executed on communication object  $y$  by process  $z$ ”.

The format used for declaring structural properties is explained and illustrated in the sample `system_file.VS` file of Figure 10. By default, if no such properties are specified, VeriSoft assumes that any process may perform any type of operation on any communication object.

## 6 Customization

The core of VeriSoft is implemented as a set of functions packaged into libraries. We now discuss these functions and how they can be used to analyze your own applications.

```

# The following options specify properties of the structure of the
# system being analyzed. These properties are used to further prune
# the state space. If any of these properties is detected to be
# violated during the search, an error is reported.
#
# (By default, if no such properties are specified, VeriSoft assumes that
# any process may perform any type of operation on any communication
# object.)
#
# Below, examples are given for processes communicating via:
# - bounded FIFO queues;
# - semaphores.
#
# NOTE! - com. objects of a given type are identified by numbers:
# *0* is the first queue (or sem) created, 1 is the second, etc.
# - processes are also identified by numbers:
# *1* is the first process created, 2 is the second, etc.

# Communication via bounded FIFO queues.

# "send_to_queue" can be executed on queue 0 by process 1:
send_to_queue 0 1

# "rcv_from_queue" can be executed on queue 0 by process 2:
rcv_from_queue 0 2

# "is_queue_full" can be executed on queue 0 by NO process:
is_queue_full 0 0

# "is_queue_full" can be executed on queue 1 by NO process:
is_queue_full 1 0

# "is_queue_empty" can be executed on queue 0 by NO process:
is_queue_empty 0 0

# "is_queue_empty" can be executed on queue 1 by NO process:
is_queue_empty 1 0

# Communication via semaphores.

# "semwait" can be executed on sem 0, index 1, by process 2:
semwait 0 1 2

# "semsignal" can be executed on sem 1, index 0, by process 1:
semsignal 1 0 1

```

Figure 10: Part 2 of sample file system\_file.VS

## 6.1 Existing VeriSoft Libraries

The list of operations intercepted by the version of VeriSoft included in the distribution package is given in the file `verisoft.h` in the directory `verisoft/bin`.

The operations on processes and communication objects are of the following types.

1. **Object creation and initialization.** These operations are executed for creating and initializing communication objects. These operations should always be executed before process creation (type 2 below), otherwise VeriSoft will report an error.
2. **Process creation.** This operation, namely the system call “fork”, creates a process. It can be executed several times to create multiple processes, or never be called in the case of a system composed of a single process. WARNING: VeriSoft assumes that a single parent process creates all the child processes!<sup>1</sup>

`nproc` in the file `system_file.VS` should be set to the number of processes in the system when the initial global state is reached. If fewer or more processes are created, VeriSoft will report an error. Note that process creation is not allowed after the initial global state is reached, i.e., after the execution of visible operations (type 3 operations below).

3. **Visible operations.** These operations are performed by processes of the system on communication objects created by operations of type 1 above. When every process in the system is about to execute its first visible operation, the whole system reaches the initial global state. Visible operations should always be executed after the first process creation operation (type 2), otherwise VeriSoft will report an error, except in the case of systems composed of a single process. Note that process creation is not allowed after the initial global state is reached.
4. **Process termination.** This operation, namely the system call “exit”, terminates the process that executes it. Processes should always terminate using this operation, otherwise VeriSoft will not detect the termination and will report a divergence instead (since the dead process is not responding anymore to VeriSoft).
5. **Object deletion.** These operations remove communication objects from the system. These operations should always be executed exclusively when the whole system terminates. For safety reasons, VeriSoft reports a warning when a process attempts to execute such an operation, and considers its execution as blocking. To avoid warnings, comment out the calls to such functions when using VeriSoft, or, even better, enclose these calls by “`#ifndef -DVERIFY`” and “`#endif`” in order to exclude them from the program when it is compiled by the script `verisoft`.

---

<sup>1</sup>Thanks to Danita Hartley for prompting this clarification.

The constraints above imply that all the processes and communication objects *visible to VeriSoft* have to be created before the system reaches its initial global state. By “hiding” the creation of processes and communication objects to VeriSoft, one can force the tool to treat a parallel composition of multiple processes as a single deterministic “black-box” process, and force it to ignore some inter-process communications. This practice is not recommended since it may be dangerous (when the behavior of the black-box process is actually nondeterministic due to internal concurrency), but it is useful for reducing the complexity of an analysis, especially in the case of very large systems (e.g., composed of tens of processes). Note that, during state-space exploration, VeriSoft records the type of visible operation performed during the execution of each transition in the state space; if, when re-executing a transition  $t$ , VeriSoft observes that the type of  $t$  is not the same as the recorded type for  $t$ , it immediately reports an error complaining that the last transition executed  $t$  is not deterministic, it prints both the recorded type and last observed type for  $t$ , and it saves the last scenario executed, whose last transition is thus  $t$ .

In the case of the examples included in the distribution package, the interception of the execution of all the types of operations listed above is implemented by mapping the names of these operations to new operations understood by VeriSoft, as specified in the file `verisoft.h`. This file is included at the beginning of the source files of the program to be analyzed. When the program is compiled, it is then linked with the appropriate VeriSoft libraries which contain executable code for the new operations. These new operations first executes some code specific to VeriSoft before calling the original corresponding operation.

In the case of a C++ program, the file `verisoft-c++.h` also needs to be included in every C++ source file, just after including `verisoft.h` as described in the previous paragraph. The file `verisoft-c++.h` includes function prototypes for the VeriSoft libraries included in the VeriSoft distribution package. After replacing “gcc” by “g++” in the script `verisoft`, one can run `verisoft` on any such C++ program exactly as described before for the case of C programs.

Note that other techniques for intercepting executions of operations could be used (such as binary-file or OS-kernel instrumentation, or the use of “wrap-up” functions intercepting events going in and out of the application being tested). This is necessary for instance when source code is not available, or cannot be recompiled. VeriSoft itself does not rely on any specific program instrumentation technique. The only constraint on the compilation of the software application to be analyzed is that, when executed under the control of the VeriSoft simulator, the source code should be compiled with the option “-g” for producing debugging information, otherwise the VeriSoft simulator will not be able to display the next line to be executed by processes in their program text.

The following miscellaneous operations are also available.

- `int VS_toss(int)` is used to simulate nondeterminism. It returns any integer between

0 and its argument. For instance, `VS_toss(1)` returns either 0 or 1. `VS_toss` introduces a branching point in the state space, with as many branches as there are possible returned values. In state-space exploration mode, all the possible branches are explored. `VS_toss` should always be executed after the first process creation operation (type 2), otherwise VeriSoft will report an error, except in the case of systems composed of a single process.

- `void VS_assert(int)` reports an assertion violation if its argument evaluates to false (zero).
- `void VS_abort(int)` does not explore the successors of the current state if its argument evaluates to false (zero). The use of this function is similar to the use of an assertion, except that no error is reported. It can be used to prune parts of the state space that are considered uninteresting by the user.
- `void VS_print(char *)` prints a one-line (no “\n”) null-terminated string (for instance, a comment) in the Trace View of the VeriSoft simulator. Inserting calls to this function inside internal computations of processes can be used to show progress in these computations in the Trace View of the VeriSoft simulator.

## 6.2 Developing New VeriSoft Libraries

The architecture of VeriSoft has been designed in a modular way in order to facilitate the integration of new libraries of communication objects. The development of new VeriSoft libraries is not discussed in this document.

## 7 Examples of the Distribution Package

All the examples included in the distribution package can be compiled and executed without the control of VeriSoft by typing (on Solaris)

```
gcc -I$VS_PATH $VS_PATH/verisoft_SunOS_5.x.o filename.c
```

where `$VS_PATH` denotes the full pathname of the `verisoft/bin` sub-directory. This command will generate an executable file `a.out`.

When you stop the execution of `a.out` (for instance by typing `Control-c`), do not forget to clean the system yourself if necessary, by using the commands `ipcs` and `ipcrm`.

## References

- [God96] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, January 1996.
- [God97] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 174–186, Paris, January 1997.