Design and Analysis of Communication Software

Part 3:

"Model-Checking" Software with VeriSoft

A New Approach to
Communication Software Analysis

Overview

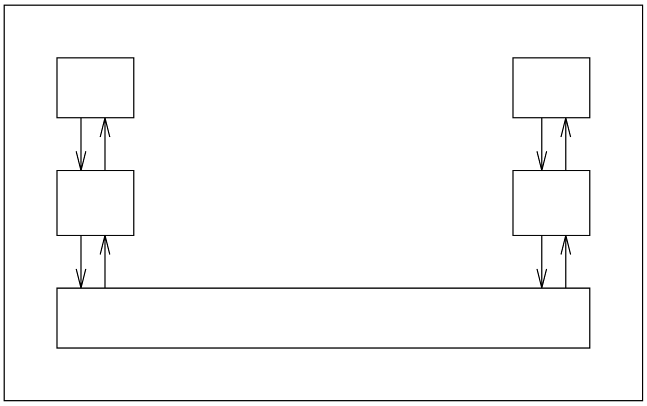
Part 3.1:

- What is VeriSoft?
- How does it work?
- Industrial applications.
- Related work and discussion.
- Project status and conclusions.

Part 3.2:

- VeriSoft demo.
- VeriSoft in practice.
- Options and features.
- Exercise.

What is VeriSoft?
Concurrent Reactive System Analysis



Each component is viewed as a "reactive" system, i.e., a system that continuously interacts with its environment.

Precisely, we assume:

- finite set of *processes* executing aribitrary code (e.g., C, C++, Java, Tcl, ...);
- finite set of *communication objects* (e.g., message queues, semaphores, shared memory, TCP connections, UDP packets,...).

Problem:
Developing concurrent reactive systems is hard!
(many possible interactions)

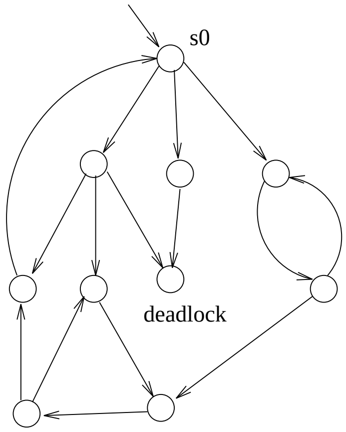
Traditional testing is of limited help! (poor coverage)

Scenarios leading to errors are hard to reproduce!

Alternative: *Systematic State-Space Exploration*

State Space (Dynamic Semantics)

- Processes communicate by executing *operations* on communication objects.
- operations on communication objects are *visible*, other operations are *invisible*;
- only executions of visible operations may be *blocking*;
- the system is said to be in a *global state* when the next operation to be executed by *every* process is *visible*;
- a move from one global state to another global state is a *transition*;
- *state space* = set of global states + transitions.



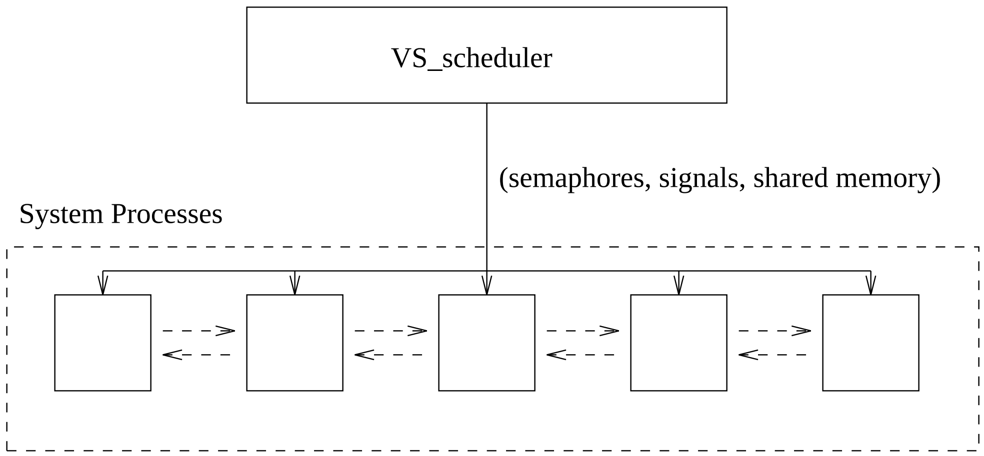
Theorem: Deadlocks and assertion violations are preserved in the “state space” as defined above.

Systematic State-Space Exploration

VeriSoft can systematically explore the state space of a concurrent reactive system.

Interceptions of all visible operations:

- control of all the processes;
- complete control over nondeterminism (i.e., concurrency + VS_toss(n));
- observation of visible operations and global states.



VeriSoft

VeriSoft searches state spaces for:

- deadlocks,
- assertion violations,
- livelocks (no enabled transition for a process during x successive transitions),
- divergences (a process does not communicate with the rest of the system during more than x seconds).

When an error is detected, VeriSoft reports a *scenario* leading to that error.

An interactive graphical simulator/debugger is also available.

How does VeriSoft work?

VeriSoft looks simple! Why did we have to wait for so long (15 years) to have it?

Existing state-space exploration tools are restricted to the analysis of *models* (i.e., abstract descriptions) of software systems.

Each state is represented by a *unique identifier*.

During state-space exploration, visited states are saved in memory (hash-table, BDD,...).

With *programming languages*, states are much more complex!

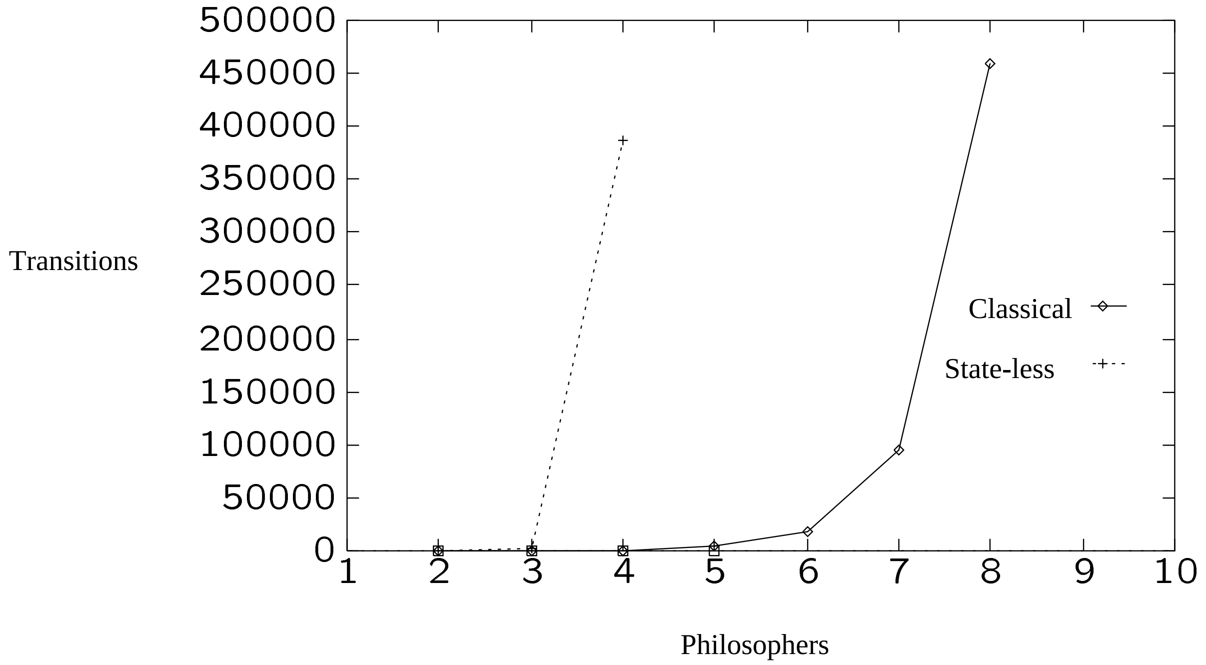
Computing and storing a "unique identifier" for each state is unrealisitc!

State-Less Search

Idea: perform a *state-less search*!
(still terminate when state space is acyclic)

Equivalent to *"state-space caching"* with an empty cache: this search technique is terribly inefficient! [H85,JJ91]

Example: dining philosophers (toy example)



For 4 philosophers, a state-less search explores 386,816 transitions, instead of 708.

Every transition is executed on average 546 times!

An Efficient State-Less Search

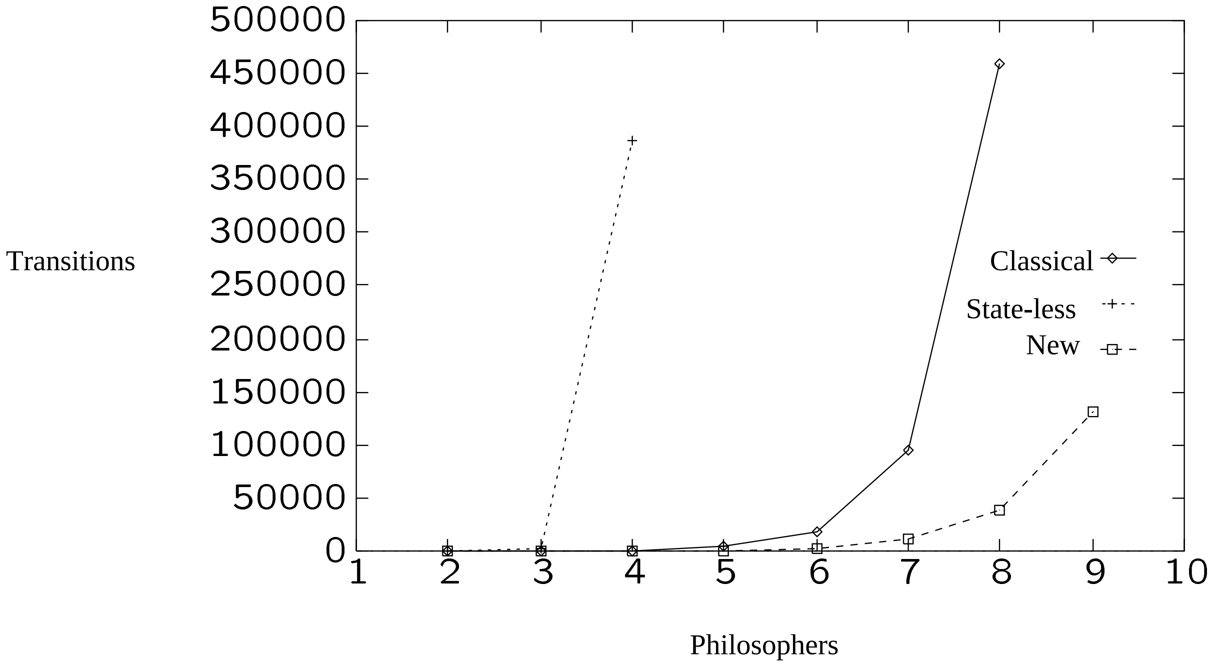
[GHP92]: Redundant explorations due to state-space caching can be strongly reduced by using *Sleep Sets* [G90], and *"partial-order methods"* in general [G96].

VeriSoft: original algorithm combining

- state-less search,
- sleep sets [G90,GW93],
- conditional stubborn sets [V90,GP93,G96].

Theorem: For finite acyclic state spaces, the above algorithm can be used for the detection of deadlocks and assertion violations without incurring the risk of any incompleteness in the verification results.

Observation:
when using this algorithm, most of the states are visited *only once* during the search.
$\leadsto$ Not necessary to store them!



VeriSoft – Summary

VeriSoft is the first tool for systematically exploring the state spaces of systems composed of several concurrent processes executing arbitrary (e.g., C or C++) code.

Originality: framework, search, tool [POPL'97].

The key to make this approach tractable is to use *smart* state-space exploration algorithms!

In practice, the search is typically incomplete.

From a given initial state, VeriSoft can always guarantee a complete coverage of the state space up to some depth.

Industrial Applications

Examples of Applications:
(within Lucent Technologies)

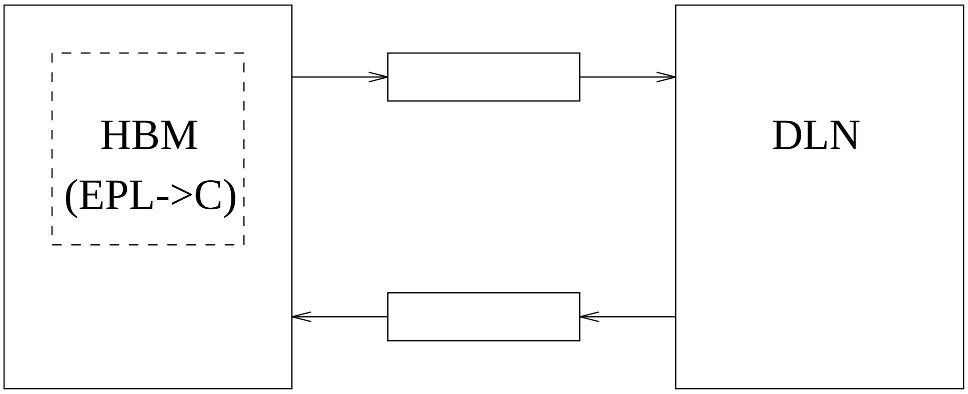
- 4ESS Heart-Beat Monitor analysis (debugging, reverse-engineering).
- Wavestar 40G integration testing (testing).
- Automatic Protection Switching analysis (interoperability protocol testing).

4ESS Heart-Beat Monitor Analysis

- May affect millions of calls per day.
- Determines status of elements connected to 4ESS switch from propagation delays of messages.
- Plays an important role in routing new calls in 4ESS switch (by triggering "No Trunk Hunt" (NTH) = switch from out-of-band to in-band signalling).
- November 1996: "field incident"...
- June 1997: calls from "field rep."...
- Code is 7 years old, modified 3 years ago.
- Several hundred lines of EPL (assembly) code.
- How does this code work exactly???

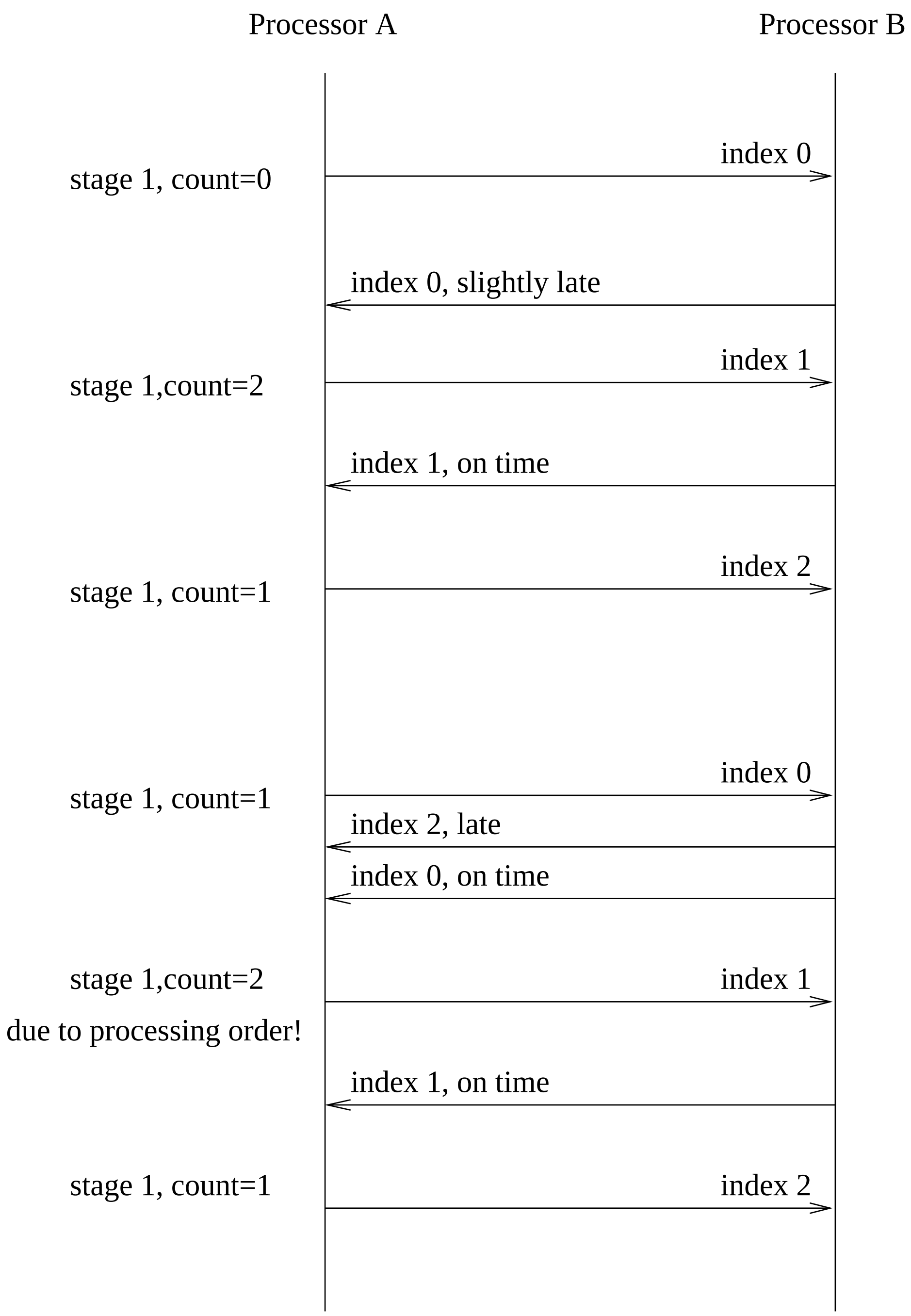
Analysis of 4ESS HBM using VeriSoft

- Translate EPL code to C code
  (using existing partial compiler).
- Build test harness for HBM C code:
  simple "wrapper" program
  (takes only a few hours!).
- Model the environment of the HBM: with
  "VS_toss(n)"
  (takes only a few hours!).
- Add "VS_assert(0)" where NTH in HBM code.
- Check properties (reverse-engineering $\leftrightarrow$ testing).



$\leadsto$ Discovered flaws in documentation and unexpected behaviors in software itself...

Example of Scenario Found



(See paper [BLTJ'98] for details.)

Conclusions of the 4ESS HBM Analysis

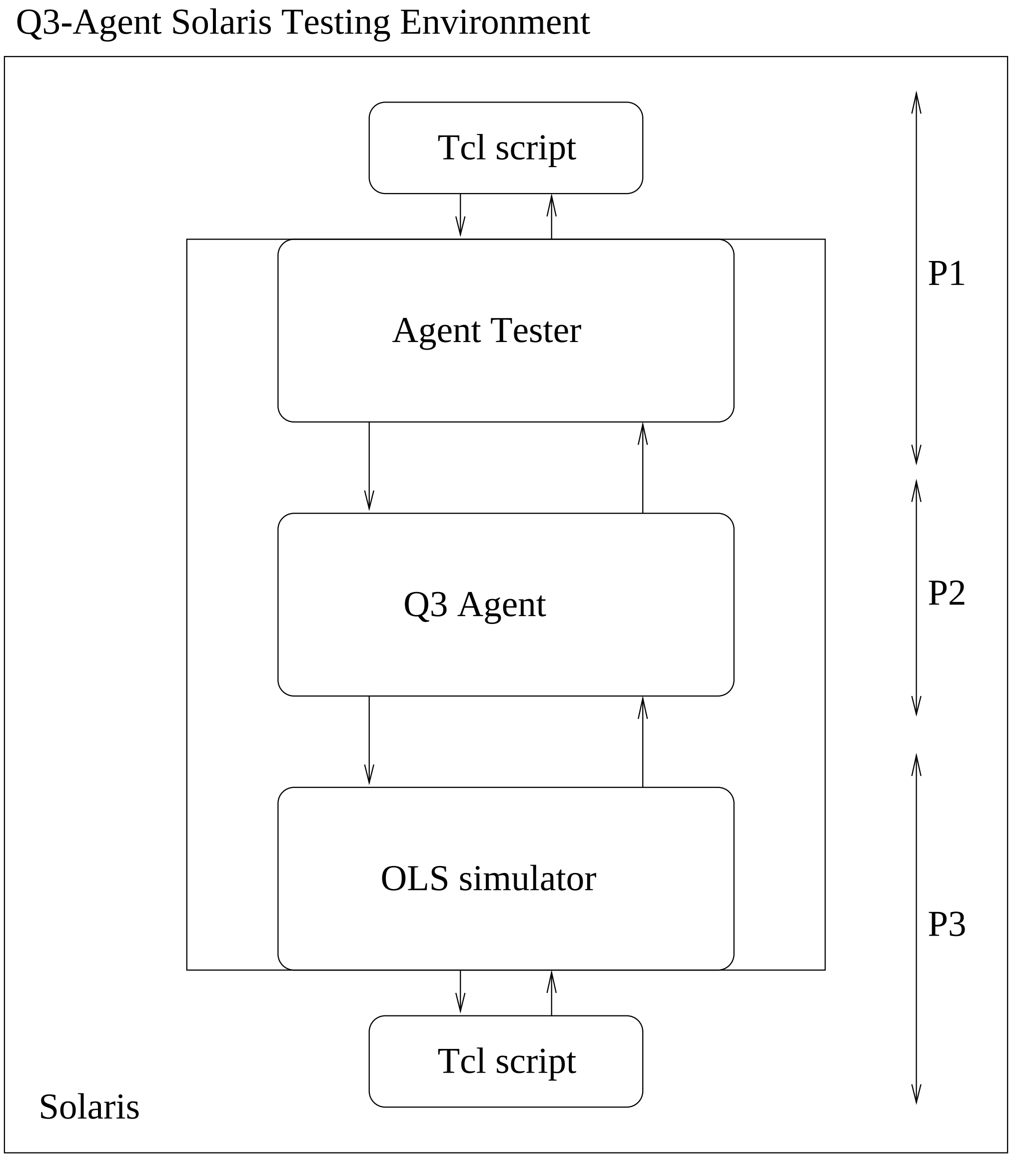
HBM: Analysis revealed flaws in documentation and unexpected behaviors in software itself.

- HBM code is very "irregular" (very hard to predict behavior).
- Similar analysis performed on previous version:
  - more sensitive, although not strictly;
  - also "irregular".
- Design of a new version:
  - passes these tests;
  - implemented in new release.

VeriSoft:

- Can quickly reveal behaviors virtually impossible to detect using conventional testing techniques (due to lack of controllability and observability).
- Strength: no need to model the application!
  - Eliminates this time-consuming and error-prone task required with other state-space exploration tools.
  - VeriSoft is WYSIWYG: great for reverse engineering!

Wavestar 40G Integration Testing



“Black-box” testing, large processes ($O(10^5 - 10^6)$ lines of C/C++ code).

Wavestar Testing with VeriSoft

- From the testers' point of view, two main new Tcl commands are available with VeriSoft:
  - VS_toss simulates nondeterminism.
  - VS_assert is used to determine whether test passed/failed.

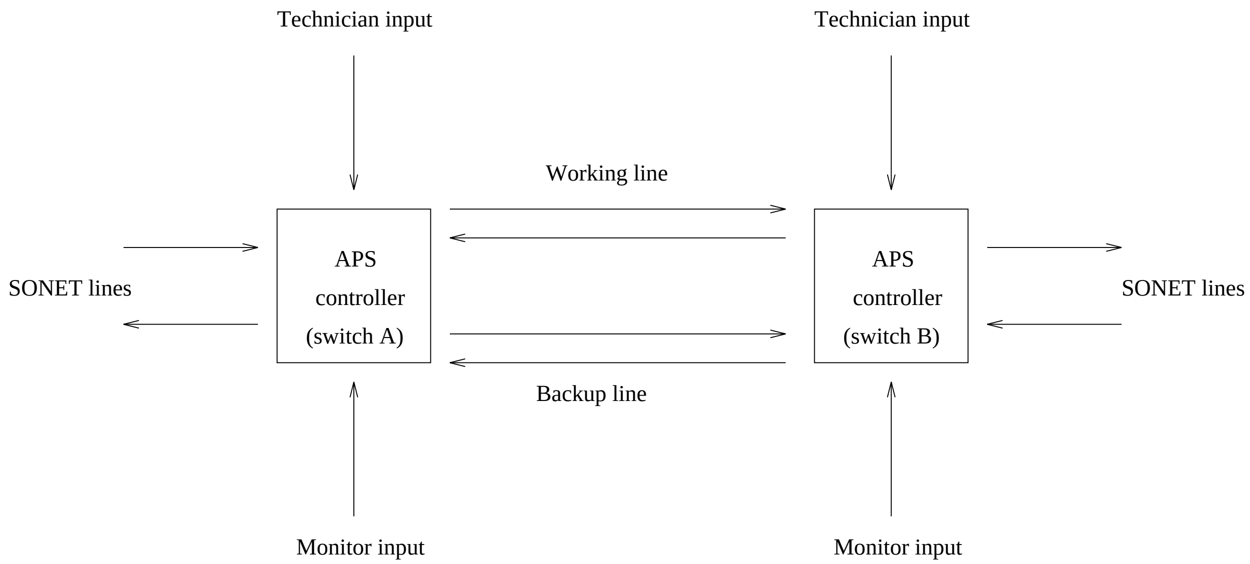
  These commands can be used anywhere (any language, any procedure, any process).
- A single nondeterministic test script can specify a family of thousands of (deterministic) test scripts.

```
[...]
if VS_toss(1) then event = MSG1
            else event = MSG2;
switch(VS_toss(2)) {
    case 0:  param = PARAM1;
        break;
    case 1:  param = PARAM2;
        break;
    case 2:  param = PARAM3;
};
send-to-IUT event param
if (test-failed) then VS_assert(0)
[...]  /* 6 possible combinations */
```

- All these test scripts are automatically generated, executed and evaluated by VeriSoft.

Testing with VeriSoft revealed significant bugs!

Automatic Protection Switching Analysis



- A 5ESS "switch-maintenance" application.
- APS protocol ensures that both switches read data from the same line. (APS is part of SONET/SDH standard.)
- Several thousands lines of C code.
- VeriSoft discovered several incompatibilities between different versions of APS code.

Comparison with Related work

Other model-checkers (for software): (e.g., SPIN, VFSMvalid)

- language dependent;
- need a model, or limited to high-level design;
- but analyzing a model is easier.

Specification-based test generation: (e.g., TestMaster)

- language dependent;
- test generation only;
- no support for concurrency
  (testing through a single interface only).

Static analysis techniques for automatic model extraction (ex of tool ?):

- language dependent + often need additional restrictions;
- abstraction is not a panacea: it always introduces unrealistic behaviors;
- overall, complementary with VeriSoft
  (e.g., see [PLDI'98]).

$\leadsto$ VeriSoft (the concept) is here to stay...

Future Work – Challenges

VeriSoft is not a panacea!

- Scalability limited by the "state explosion" problem...
- Used naively, poor feedback is likely, but used properly, can be extremely effective.
  $\leadsto$ Training is necessary!
- Help to model the environment...
  "Automatically Closing Open Reactive Programs" [PLDI'98]
- Improve feedback to user...
  coverage information, state-space visualization
- Checking properties on partial state spaces... [CAV'99]
- Exploiting symmetry (e.g., client-server applications)... [FORTE-PSTV'99]
- etc.

References: see `http://www.bell-labs.com/~god`

VeriSoft Project Status

Research prototype under development since 1996.

Used in several Lucent BU's over the past 2 years.

Application domains: switch maintenance, network management, call processing,...

Products: voice/data switches, optical networking, wireless,...

Available also outside Lucent since January 1999 (this version runs on Solaris and Linux, and simulator supports C/C++).

Possible New Venture...

Visit the VeriSoft web-site!

`http://www.bell-labs.com/projects/verisoft/`

Summary

VeriSoft is a tool for *systematically* testing concurrent/reactive software.

It can quickly reveal behaviors that are virtually impossible to detect using conventional testing techniques.

Benefits:

- Detection of hard to pinpoint bugs and gain of confidence about correctness (by increasing test coverage by several orders of magnitude).
- Reduction of development intervals and costs (by catching bugs earlier and avoiding major product failures).

VeriSoft finds bugs that would otherwise be found by the customer!

Conclusion

Claim: VeriSoft (tool + approach) can find bugs in any nontrivial concurrent/reactive software system.

1. Concurrent/reactive/real-time software is hard to design and test.
2. Traditional testing techniques are not adequate (poor coverage, lack of observability and controlability).
3. Systematic testing using an approach specially developed for detecting race conditions and timing issues can rather easily expose previously unknown bugs.

So the real question is:
How much ($) do you care about bugs?

VeriSoft Demo

(No slides)

VeriSoft in Practice

Step 1: Setup

- Instrumentation:
  specify the application's testing interface (done only once).
- Test Harness:
  model the application's environment, possibly using multiple processes and nondeterminism (Ex: VS_toss(1)).

Step 2: Operation Modes

- Manual Simulation:
  interactive exploration of scenarios by the user.
- Automatic Simulation:
  automatic search for errors.
- Guided Simulation:
  replay and examine scenarios leading to errors.

Note: the user defines the state space, VeriSoft explores it.

Using VeriSoft: Options and Features

Note: instructions for the Solaris/C version of VeriSoft 2.0.x only.

First, add "~patrice1/verisoft/bin" to your environment variable PATH (in `.cshrc` at Stanford).

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.

`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.

Always run VeriSoft from a directory you can write to.

VeriSoft always expects to find in the current directory two files:

- `system_file.VS` is used to specify search parameters.
- `a.out` is the main executable file for the application being tested.

A Word of Caution

VeriSoft and the program it analyzes use Unix IPC resources such as semaphores and message queues.

IPC resources are not freed automatically when program terminates or even when logging out!

BE CAREFUL!!!

Two main Unix commands:

- `ipcs` reports IPC resources currently used on a machine.
- `ipcrm` (+options) removes IPC resources from a machine.

See man pages for these commands.

Note: VeriSoft keeps track of "visible" objects and recycles these automatically.

Note: `~patrice1/verisoft/bin/ipclear` is a script using `ipcs` and `ipcrm` for removing all the IPC resources on a machine.

VeriSoft Libraries (`verisoft.h`)

In the version of VeriSoft installed at Stanford, interception of system calls is implemented via source code instrumentation (using `verisoft.h`) and linkage to VeriSoft libraries.

System calls intercepted are of 5 types:

1. **Object creation and initialization** (ex: get_bounded_queue; executed before type 2)
2. **Process creation** (ex: fork; executed before type 3)
3. **Visible operations** (ex: send_to_queue)
4. **Process termination** ("exit" must be used, otherwise divergence)
5. **Object deletion** (ex: remove_queue; can be omitted, see manual)

The file `verisoft.h` contains a predefined set of types of communication objects (semaphores and message queues) using the functions defined in the files `semdef.h` and `msgdef.h` of the verisoft/bin directory.

Note: `#ifndef -DVERIFY` bypasses the instrumentation.

Other Visible Operations

The following miscellaneous operations are also available.

- `int VS_toss(int)` (ex: `VS_toss(1)` returns either 0 or 1)
- `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).
- `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.

Search Parameters (system_file.VS)

```
#
# 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)
```

Structural Properties (system_file.VS)

```
# 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.)
#
# 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_empty" can be executed on queue 0 by NO process:
is_queue_empty 0 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
```

Exercise

Go to
`~patrice1/verisoft/examples/ac-controller`.

Copy all the files there to a fresh directory where you have read and write permissions.

Go to that directory.

Run verisoft with "`verisoft main.c`".

An assertion violation is then detected.

Try to fix the problem, and then run VeriSoft again to test whether your solution is correct.

(A solution is given in the file "main2.c".)

Need help? See the VeriSoft Reference Manual or the section [Demo] of the VeriSoft web-site.

Main References

On VeriSoft:

"Model Checking for Programming Languages using VeriSoft", P. Godefroid, POPL'97.

On Partial-Order Methods:

"Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem", P. Godefroid, LNCS 1032.

On the VeriSoft tool (current implementation):

See the VeriSoft Reference Manual `/tmp/manual.ps` or `~patrice1/verisoft/manual.ps.Z` on saga3 at Stanford.

See also the VeriSoft web-site:
`http://www.bell-labs.com/projects/verisoft/`

On Applications:

"Model Checking Without a Model: An Analysis of the Heart-Beat Monitor of a Telephone Switch using VeriSoft", P. Godefroid, B. Hanmer and L. Jagadeesan, ISSTA'98. Journal version in Bell Labs Tech. Journal, 1998.

See `http://www.bell-labs.com/~god`