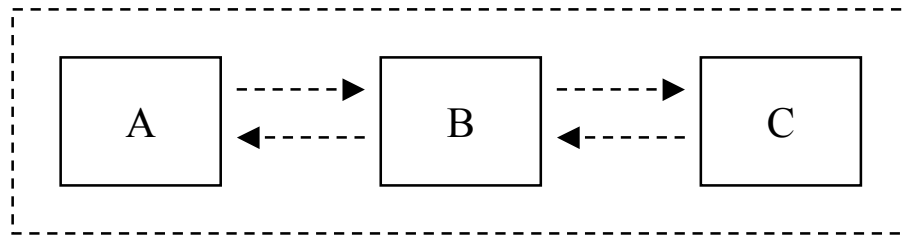

“Model Checking” Software with

VeriSoft

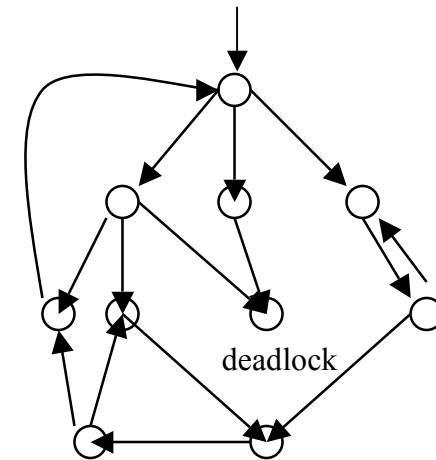
Patrice Godefroid

Bell Laboratories, Lucent Technologies

“Model Checking”



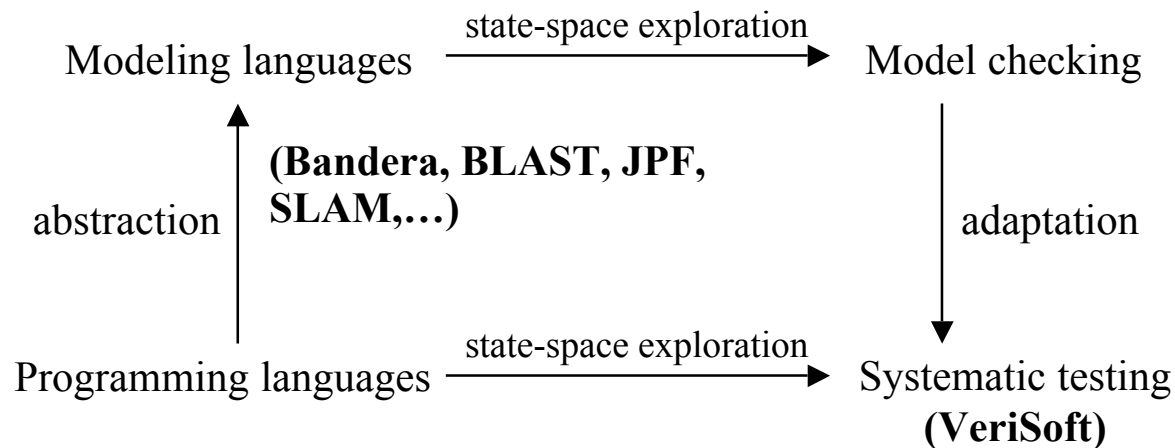
Each component is modeled by a FSM.



- Model Checking = systematic state-space exploration = exhaustive testing.
- “Model Checking” = “check whether the system satisfies a temporal-logic formula”.
 - Example: $G(p \rightarrow Fq)$ is an LTL formula.
- Simple yet effective technique for finding bugs in high-level hardware and software designs (examples: FormalCheck for Hw, SPIN for Sw, etc.).
- Once thoroughly checked, models can be compiled and used as the core of the implementation (examples: SDL, VFSM,...).

Model Checking of Software

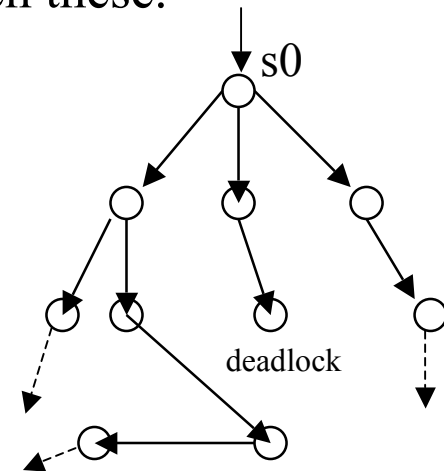
- Challenge: how to apply model checking to analyze **software**?
 - “Real” programming languages (e.g., C, C++, Java),
 - “Real” size (e.g., 100,000’s lines of code).
- Two main approaches to software model checking:



VeriSoft: Systematic State-Space Exploration

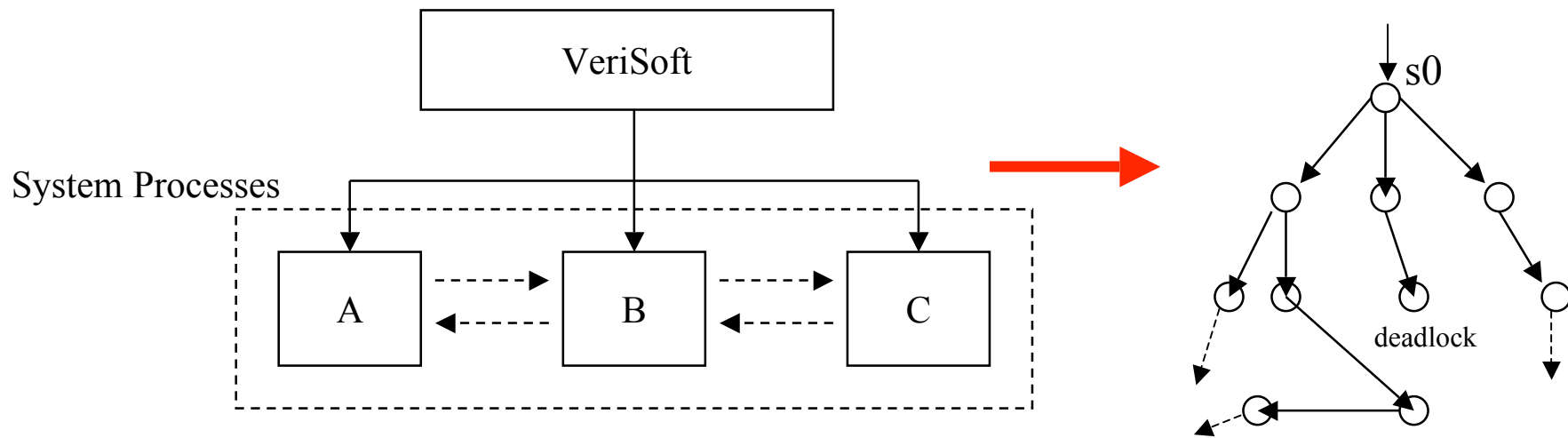
- State Space (Dynamic Semantics)= “product of (Unix) processes”
 - Processes communicate by executing operations on com. objects.
 - Operations on com. objects are visible, other operations are invisible.
 - Only executions of visible operations may be blocking.
 - The system is in a global state when the next operation of each process is visible.
 - State Space = set of global states + transitions between these.

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



VeriSoft

- Controls and observes the execution of concurrent processes of the system under test by intercepting system calls (communication, assertion violations, etc.).
- Systematically drives the system along all the paths (=scenarios) in its state space (=automatically generate, execute and evaluate many scenarios).
- From a given initial state, one can always guarantee a complete coverage of the state space up to some depth.
- Note: analyzes “closed systems”; requires test driver(s) possibly using “VS_toss(n)”.



VeriSoft State-Space Search

- Automatically searches for:
 - deadlocks,
 - assertion violations,
 - divergences (a process does not communicate with the rest of the system during more than x seconds),
 - livelocks (a process is blocked during x successive transitions).
- A scenario (=path in state space) is reported for each error found.
- Scenarios can be replayed interactively using the VeriSoft simulator (driving existing debuggers).

The VeriSoft Simulator

The screenshot displays the VeriSoft Simulator interface with several overlapping windows:

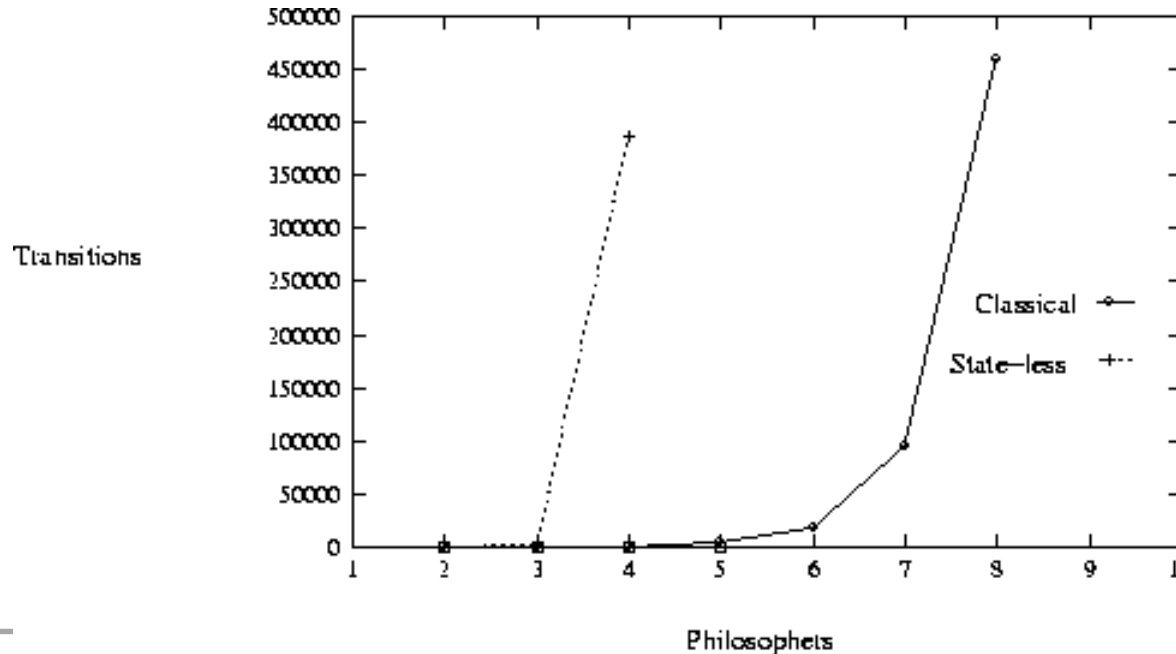
- VeriSoft Simulator - Trace View:** Shows a sequence of events for Process 1 and Process 2. Process 1 sends a message to Process 2, and Process 2 receives it. The trace shows the state of the system at each step.
- VeriSoft Simulator - (Pruned) State Space View:** Displays a state space graph. The graph shows the sequence of states reached during the simulation. The initial state is highlighted, and the current state is also marked. The graph shows the sequence of states reached during the simulation.
- VeriSoft Simulator:** A central window showing the current state of the simulation. It displays the state of the system and the current process being executed. It also shows an "Assertion violation!" message.
- VeriSoft Simulator - Process 1:** Shows the source code for Process 1. The code includes a loop that checks for a door being closed and a test for a room being hot. The current state of the process is highlighted.
- VeriSoft Simulator - Process 2:** Shows the source code for Process 2. The code includes a loop that checks for a door being closed and a test for a room being hot. The current state of the process is highlighted.
- VeriSoft Simulator - State Space Filter:** A window for filtering the state space. It contains a text regular expression field and a list of labels and processes. The filter is currently set to show only states where the door is closed.

Originality of VeriSoft

- VeriSoft is the first model checker for software systems described in general-purpose programming languages such as C and C++ [POPL97].
- VeriSoft looks simple! Why wasn't this done before?
- Previously existing state-space exploration tools:
 - restricted to the analysis of models of software systems;
 - each state is represented by a unique identifier;
 - visited states are saved in memory (hash-table, BDD,...).
- With programming languages, states are much more complex!
- Computing and storing a unique identifier for every state is unrealistic!

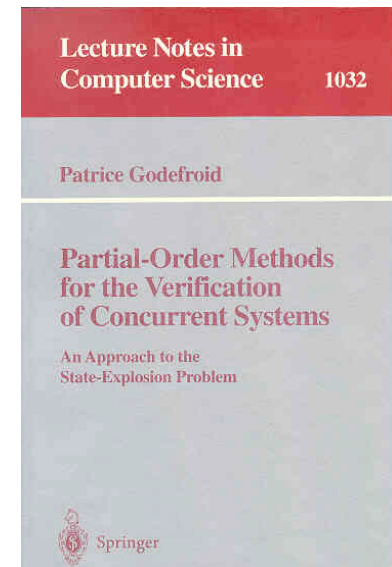
“State-Less” Search

- Don’t store visited states in memory: still terminates when state space is finite and acyclic... but terribly inefficient!
- 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!



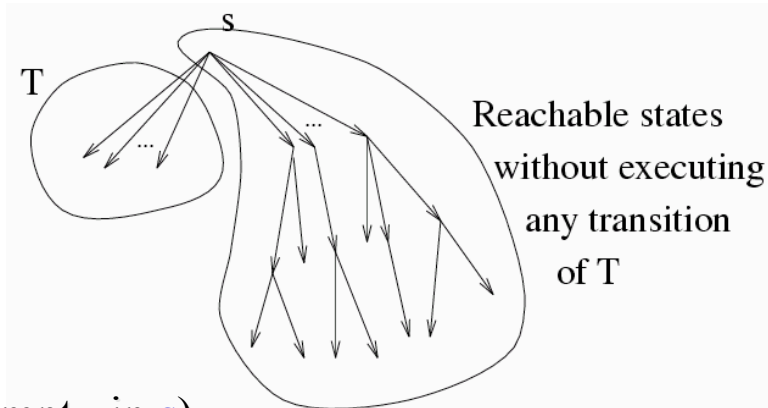
Partial-Order Reduction in Model Checking

- A state-less search in the state space of a concurrent system can be much more efficient when using “partial-order methods”.
- POR algorithms dynamically prune the state space of a concurrent system by eliminating unnecessary interleavings while preserving specific correctness properties (deadlocks, assertion violations,...).
- Two main core POR techniques:
 - Persistent/stubborn sets (Valmari, Godefroid,...)
 - Sleep sets (Godefroid,...)

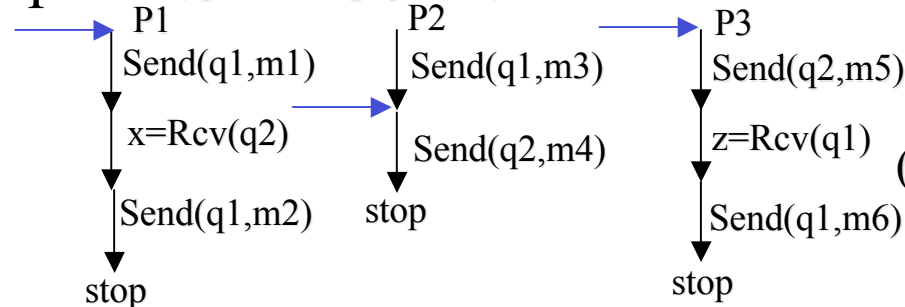


Persistent/Stubborn Sets

- Intuitively, a set T of enabled transitions in s are persistent in s if whatever one does from s *while remaining outside of T* does not interact with T .



- Example: ($q1$ is empty in s)



$\{P1:Send(q1,m1)\}$ is persistent in s

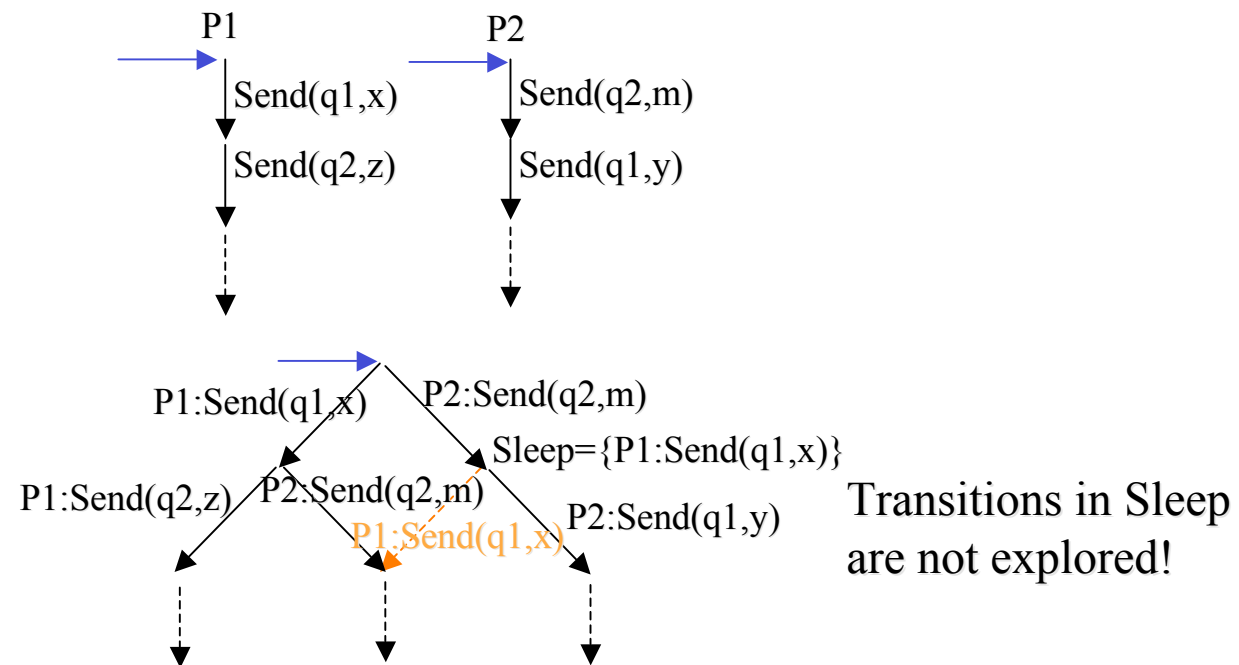
The most advanced algorithms for (statically) computing persistent sets are based on “stubborn sets”
[Valmari]

- Limitation: need info on (static) system structure.
 - VeriSoft only exploits info on next transitions and “system_file.VS”.

Sleep Sets

- Sleep Sets exploit local independence (commutativity) among enabled transitions. One sleep set is associated with each state.

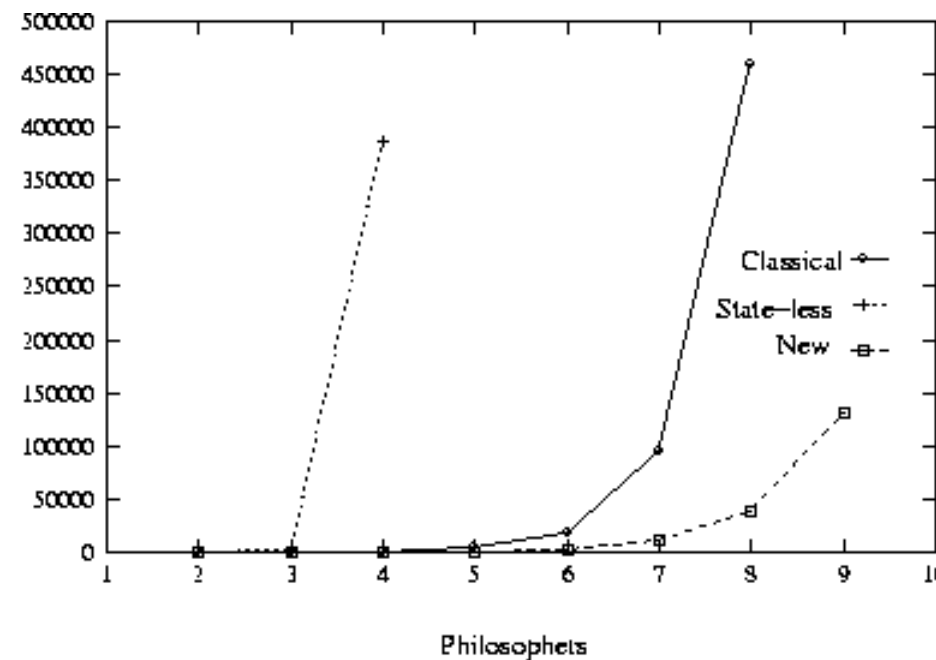
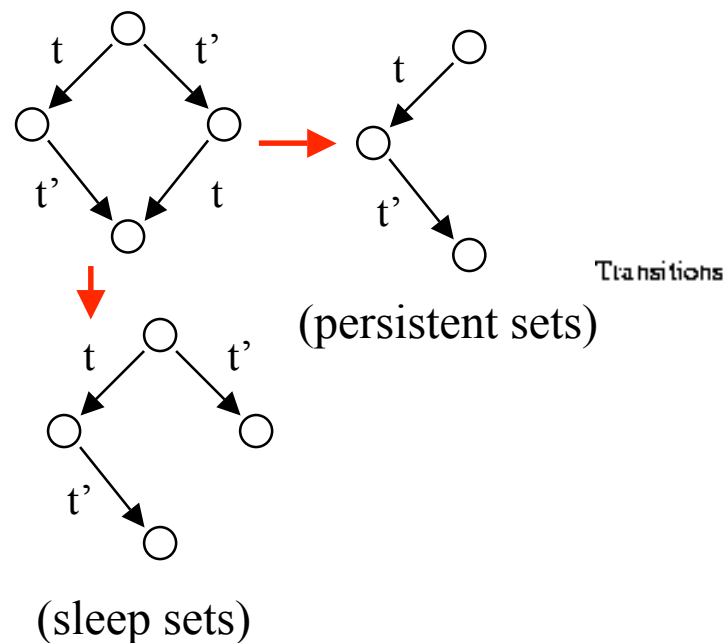
- Example:



- Limitation: alone, no state reduction.
 - Sleep sets are easy to implement in VeriSoft since they only require information on next transitions.

An Efficient State-Less Search

- With POR algorithms, the pruned state space looks like a tree!
- Thus, no need to store intermediate states!



- Without POR algorithms, a state-less search in the state space of a concurrent system is untractable.

VeriSoft - Summary

- Two key features distinguish VeriSoft from other model checkers
 - Does not require the use of any specific modeling/programming language.
 - Performs a state-less search.
- Use of partial-order reduction is key in presence of concurrency.
- 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.

Users and Applications

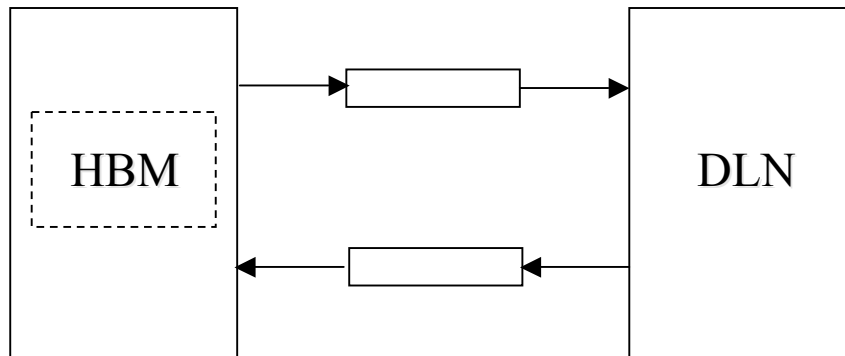
- Development of research prototype started in 1996.
- VeriSoft 2.0 available outside Lucent since January 1999:
 - 100's of licenses in 25+ countries, in industry and academia
 - Free download at <http://www.bell-labs.com/projects/verisoft>
- Examples of applications in Lucent:
 - 4ESS HBM unit testing and debugging (telephone switch maintenance)
 - WaveStar 40G R4 integration testing (optical network management)
 - 7R/E PTS Feature Server unit and integration testing (voice/data signaling)
 - CDMA Cell-Site Call Processing Library testing (wireless call processing)

Application: 4ESS HBM [ISSTA98]

- 4ESS switches control millions of calls every day.
- Heart-Beat Monitor (HBM) determines the status of elements connected to 4ESS switch by monitoring propagation delays of messages to/from these elements.
- HBM decides how to route new calls in 4ESS switch (i.e., decides to switch from out-of-band to in-band signaling - called NTH).
- November 1996: “field incident”; June 1997: 2nd field incident...
- HBM code = 100s of lines of EPL (assembly) code, 7/3 years old
- Hoes does this code work exactly???

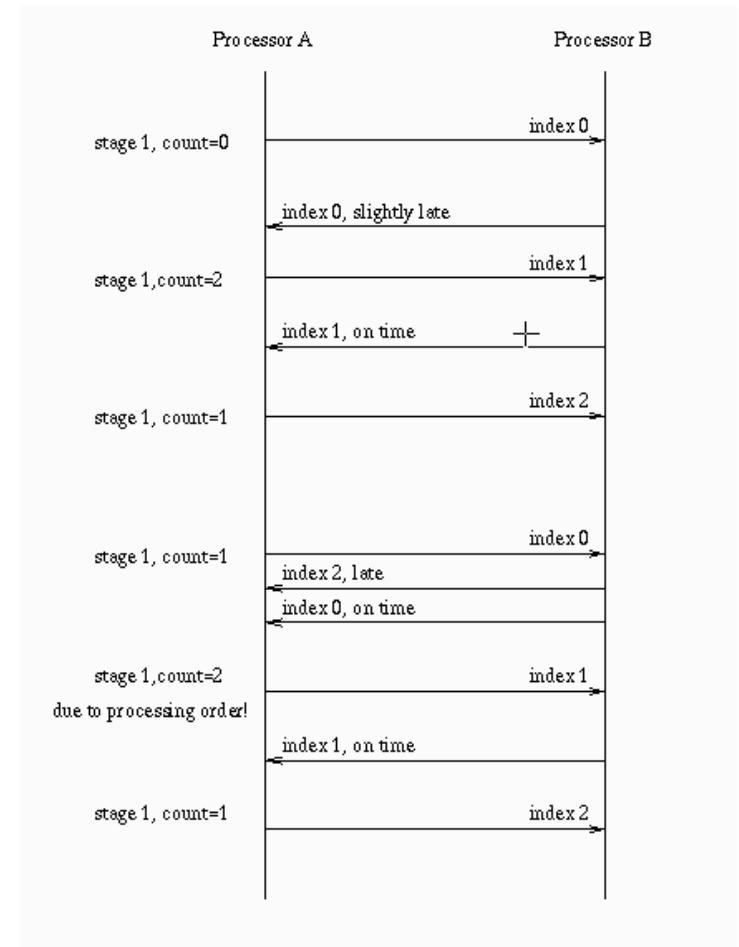
Application: 4ESS HBM (continued)

- Translate EPL code to C code (using existing partial translator)
- Build test harness for HBM C code, model its environment (using “VS_toss(n)”), add “VS_assert(0)” where HBM code hits NTH (took only a few hours!)



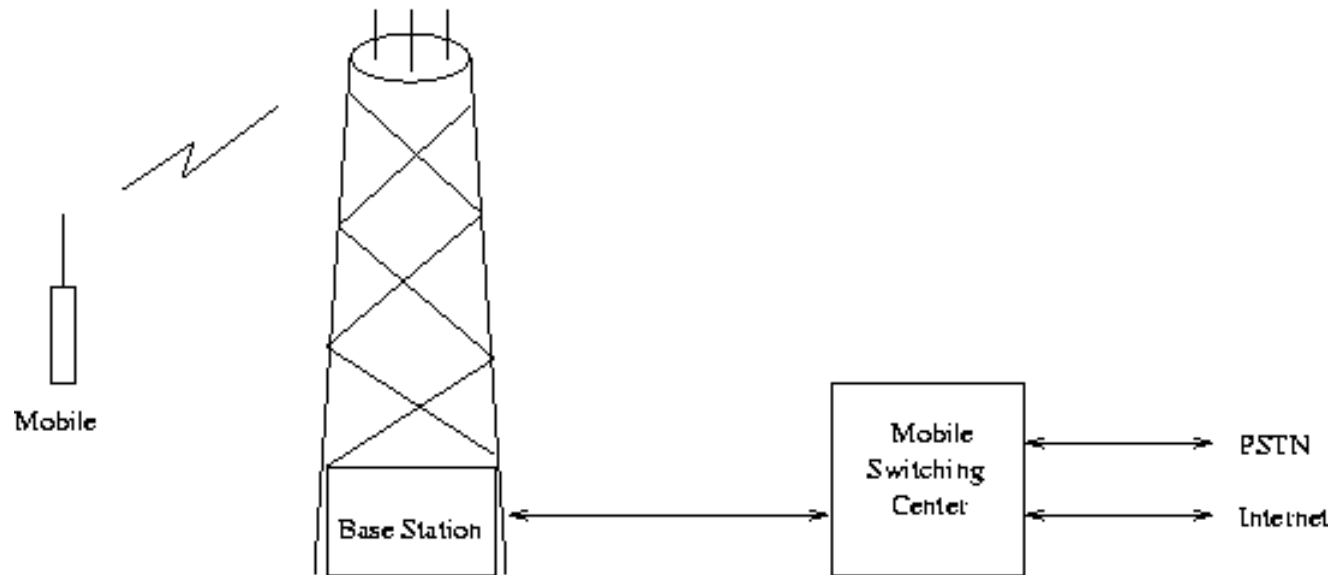
- Check properties (reverse eng./testing)
- Discovered several flaws in software and its documentation... [ISSTA98]

Example of scenario found:



Application: CDMA Base Station SW [ICSE02]

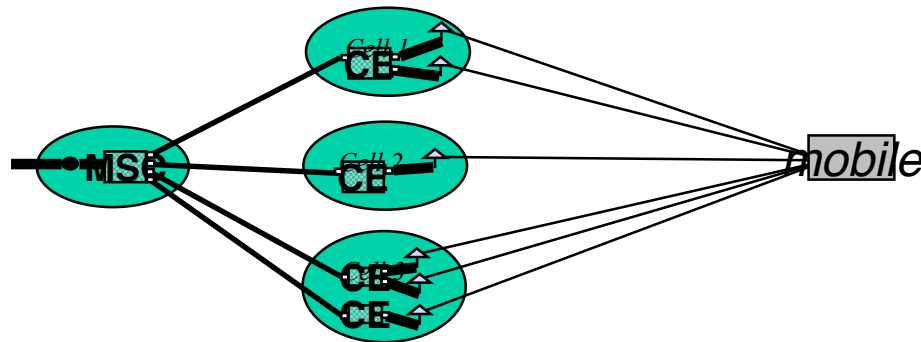
- CDMA wireless network infrastructure is a multi-billion dollar market (Lucent = #1).
- Three main components of a wireless network:



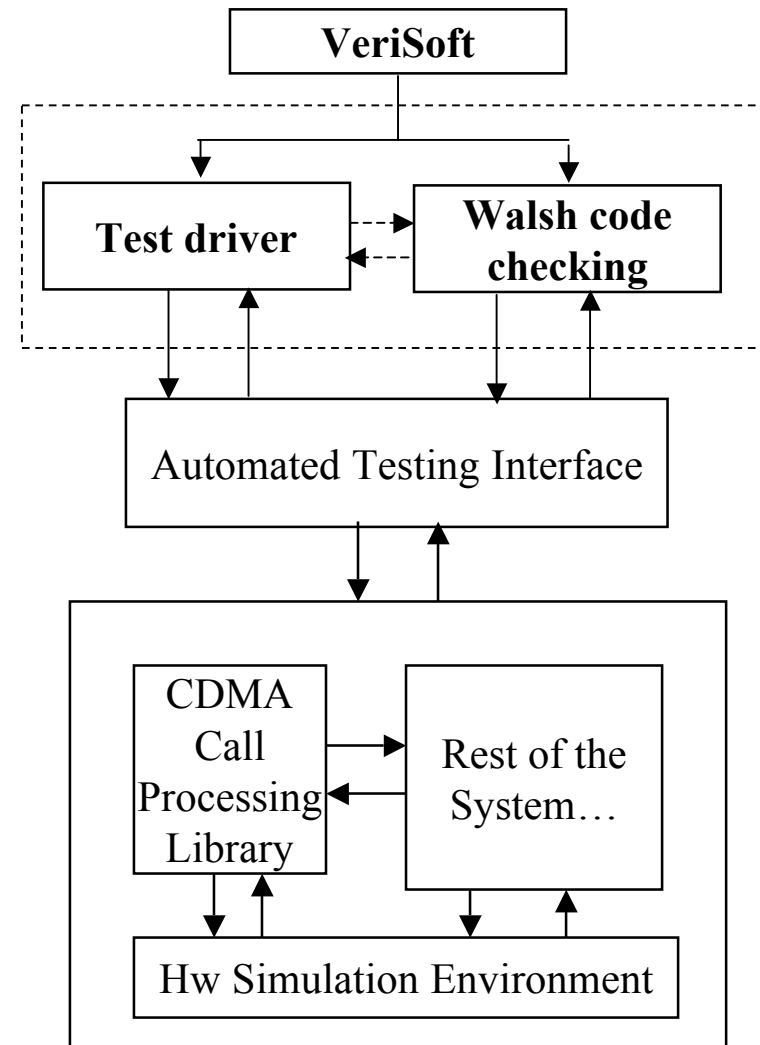
- CDMA is becoming the standard for air interface (vs. TDMA).
 - Same band of RF spectrum shared by many mobiles (using “Walsh codes”)

Application: CDMA Base Station SW (continued)

- CDMA Base Station Call-processing software library involves complex dynamic resource-allocation algorithms and handoffs scenarios (100,000's lines of C/C++ code).



- How to test reliably this software? VeriSoft
 - Increased test coverage from $O(10)$ to $O(1,000,000)$ scenarios.
 - Automatic regression testing for multiple cell-sites and releases (more than 1,500 VeriSoft runs in 2000-2001).
 - Found several critical bugs...[ICSE2002]



Discussion: Strengths of VeriSoft

- Used properly, very effective at finding bugs
 - can quickly reveal behaviors virtually impossible to detect using conventional testing techniques (due to lack of controllability and observability)
 - compared with conventional model checkers, no need to model the application!
 - Eliminates this time-consuming and error-prone step
 - VeriSoft is WYSIWYG: great for reverse-engineering
- Versatile: language independence is a key strength in practice
- Scalable: applicable to very large systems, although incomplete
 - the amount of nondeterminism visible to VeriSoft can be reduced at the cost of completeness and reproducibility (not limited by code size)

Discussion: Limitations of VeriSoft

- Requires test automation:
 - need to run and evaluate tests automatically (can be nontrivial)
 - if test automation is already available, getting started is easy
- Need be integrated in testing/execution environment
 - minimally, need to intercept VS_toss and VS_assert
 - intercepting/handling communication system calls can be tricky...
- Requires test drivers/environment models (like most MC)
- Specifying properties: the more, the better... (like MC)
 - Restricted to safety properties (ok in practice); use Purify!
- State explosion... (like MC)

Discussion: Conclusions

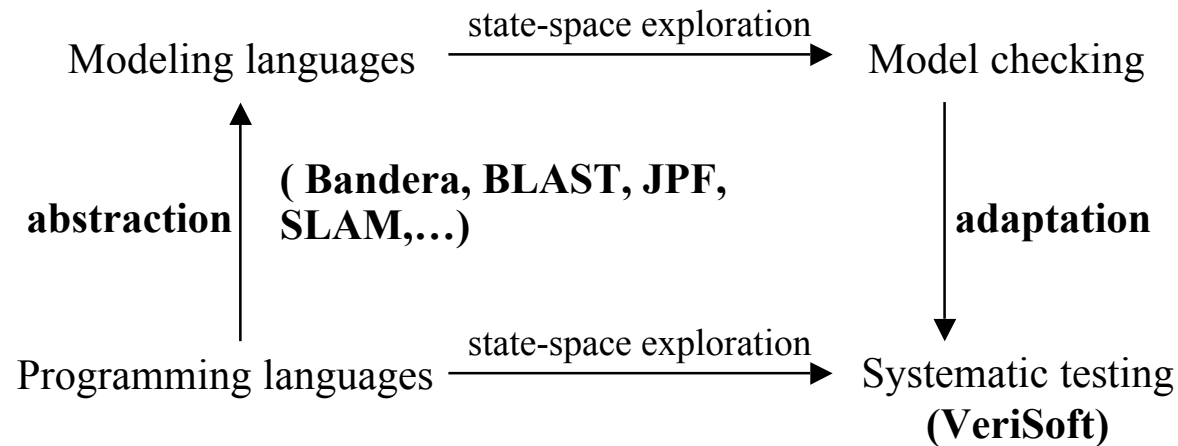
- VeriSoft (like model checking) is not a panacea.
 - Limited by the state-explosion problem,...
 - Requires some training and effort (to write test drivers, properties, etc.).
 - “Model Checking is a push-button technology” is a myth!
- Used properly, VeriSoft is very effective at finding bugs.
 - Concurrent/reactive/real-time systems are hard to design, develop and test.
 - Traditional testing is not adequate.
 - “Model checking” (systematic testing) can rather easily expose new bugs.
- These bugs would otherwise be found by the customer!
- So the real question is “How much (\$) do you care about bugs?”

Comparison with Related Work

- Traditional model checkers: (e.g., SPIN, SDLvalid, etc.)
 - language dependent,
 - requires a model or limited to high-level design,
 - but analyzing a model is easier.
- Specification-based test generation: (e.g., TestMaster, etc.)
 - language dependent,
 - test generation only,
 - no support for concurrency.
- Software model checkers based on static analysis and abstraction:
 - see next slide.

Model Checking of Software

- Two complementary approaches to software model checking:



Automatic Abstraction (static analysis):

- Idea: parse code to generate an abstract model that can be analyzed using model checking.
- No execution required but language dependent.
- Produce spurious counterexamples (unsound).
- Can prove correctness (complete).

Systematic Testing (dynamic analysis):

- Idea: control the execution of multiple test-drivers/processes by intercepting systems calls.
- Language independent but requires execution.
- Counterexamples arise from code (sound).
- Provide a complete state-space coverage up to some depth only (incomplete).

VeriSoft Project: Related Work

- First paper on VeriSoft: [POPL97]
- Examples of related research issues: (joint work with many others!)
 - How to automatically “close” open reactive programs? [PLDI98]
 - How to automatically synthesize a spec from dynamic observations? [TACAS97]
 - How to analyze effectively partial state-spaces?
[CAV99,CONCUR00,CONCUR01,CAV02,VMCAI03,EMSOFT03,LICS04...]
 - How to exploit symmetry (e.g., as in client-server applications)? [PSTV-FORTE99]
 - How to test systems without ever writing a test driver? [FSE2000]
 - VeriWeb: automatically testing dynamic websites [WWW2002]
 - How to explore very large state spaces using genetic algorithms? [TACAS2002]
 - Etc. See my web-page (www.bell-labs.com/~god) for complete references.
- Current and future work:
 - automatic generation of (nondeterministic) test drivers from static analysis...
 - goal: more users for VeriSoft by making unit testing a reality in the sw industry!

Conclusions

- VeriSoft is a tool for systematically testing concurrent/reactive sw
- Computes the “product” of OS processes with run-time scheduler
- Is language independent (C, C++,...): no static analysis
- Performs a state-less search and makes heavy use of partial-order reduction algorithms when concurrency
- Can provide full state-space coverage but typically up to some depth only; first (“bounded”) model checker for C/C++/etc.
- 100’s of non-commercial licenses in industry and academia
- Free download at <http://www.bell-labs.com/projects/verisoft>