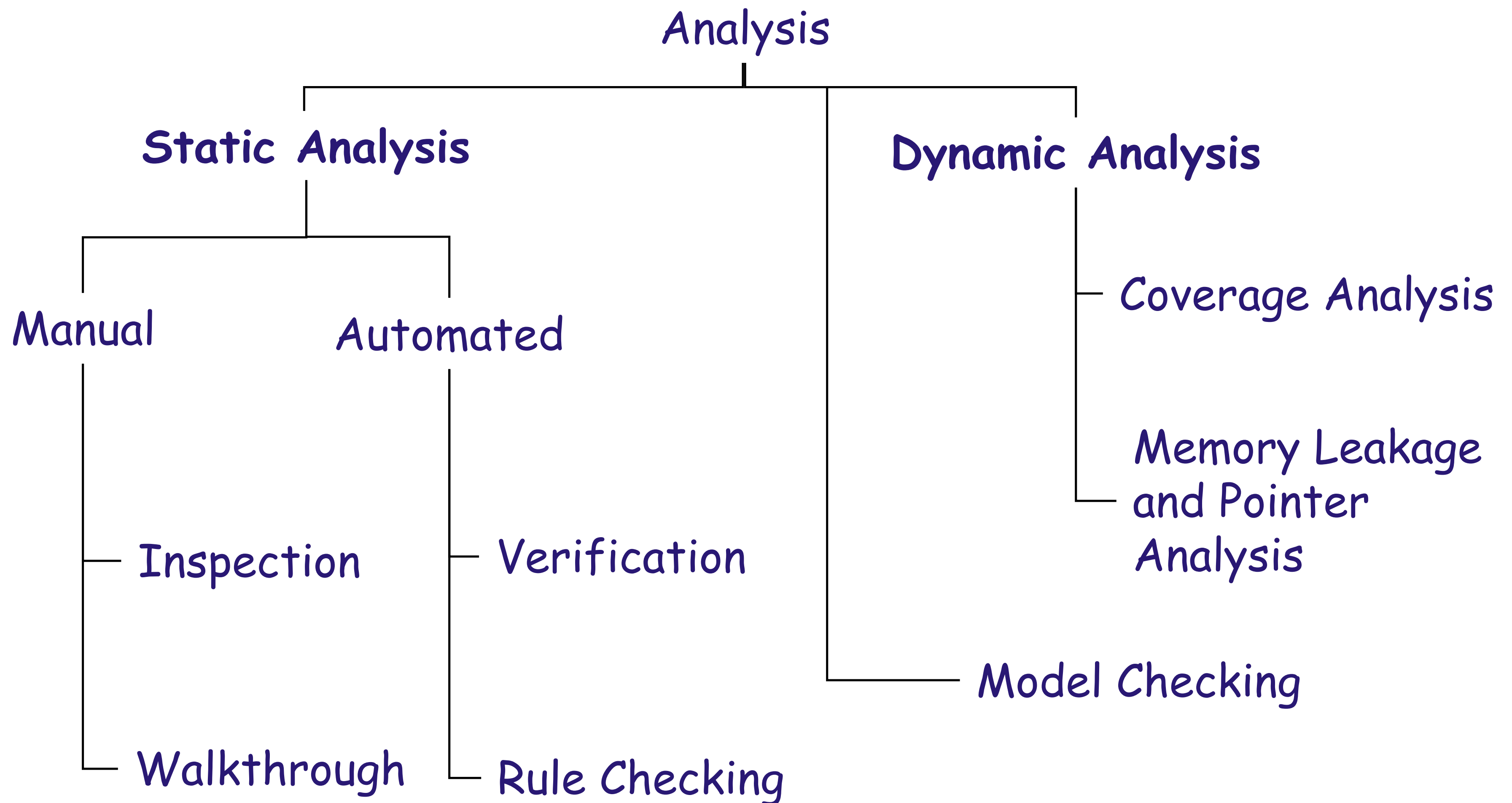
Analysis
Tools & Techniques

Taxonomy



Dynamic Tools

- Purify (formerly Rational, now owned by IBM)
- Insure++ (Parasoft)
- Valgrind (free)
- Helgrind (free)
- Time Rover (Temporal Solutions)

Purify Features

- Errors Detected
  - array bounds errors
  - accesses through dangling pointer
  - uninitialized memory read
  - memory allocation errors
  - memory leaks
- Filtering and detail level customization
- GUI and command line interface
- Integrated with VC++, ClearQuest, TestFactory

Runtime Debugger Technology

- Two main techniques:
  - Object Code Insertion (used by Rational Purify)
    - Instrumentation of the code happens on the **executable** file.
    - Involves the instrumentation of each object file and library by adding special instructions to every store and read instruction.
    - Allows analysis ***even if you don't have the source code.***
  - Source Code Insertion (used by Rational Test RealTime)
    - Instrumentation of the code happens on the source file
    - Involves parsing, analyzing and converting the original code into a new equivalent source code. Then compile the new code.
    - Generally more accurate and has access to more information about the program pointers and memory blocks. Can detect errors at compile time also.

Array bounds errors

- **Purify reports Array Bounds Read and Array Bounds Write.**
- Program writes memory past the bounds of an allocated block.
- Memory corrupted may be used much later in the program.
- Symptom could be far away from the cause.
- Program may behave in an unpredictable way, dependent upon the values accessed in the memory block corrupted.

Accesses through dangling pointer

- **Purify reports Free Memory Read and Free Memory Write.**
- Access to a freed block of memory
- Access can be a Read or a Write
- Outcome of program is unpredictable:
  - if memory block not reallocated, expected value might still be there, and program seems OK.
  - if memory block reallocated program might fail, dependent on timing issues, memory allocation pattern, etc.

Uninitialized memory read

- **Purify reports Uninitialized Memory Read.**
- Read to an allocated block of memory not yet initialized.
- Outcome of program is unpredictable, depending on the context.
- If memory contains value remaining from previous memory use, program will behave erratically.

Memory allocation errors

- **Purify reports Freeing Mismatched Memory and Freeing Invalid Memory.**
- Memory allocated in C++ with **new** but freed with **free** (instead of **delete**) may cause corruption of heap structures.
- If these structures are accessed, program can fail.
- Attempt to free a local variable on the stack might succeed and corrupt local variables or function return.

Memory leaks

- **Purify reports Memory Leaks**
- Memory allocated but **never freed**, and for which no pointers exist.
- Memory blocks cannot be used nor freed, and occupy address space.
- Performance of the program degrades and eventually it fails from lack of memory

Purify

Microsoft Development Environment [design] - Scribble3.pcy Data Browser: Purify'd Scribble.exe

File Edit View Project Build Debug Tools PurifyPlus Window Help

Debug

Tool... | Data Modeler | Clipboard R... | General | Pointer

Scribble3.pcy D...y'd Scribble.exe

Error View

```
Loading Purify'd ADVAPI32.dll at 0x67dd0000
Loading Purify'd KERNEL32.dll at 0x67e60000
Loading Purify'd NTDLL.dll at 0x67f50000
Loading Purify'd RPCRT4.dll at 0x68000000
Loading SHLWAPI.dll at 0x70a70000
Loading Purify'd OLEACC.dll at 0x74c80000
Loading Purify'd OLE32.dll at 0x771b0000
Loading GDI32.dll at 0x77c70000
Loading USER32.dll at 0x77d40000
Loading ADVAPI32.dll at 0x77dd0000
Loading kernel32.dll at 0x77e60000
Loading ntdll.dll at 0x77f50000
Loading RPCRT4.dll at 0x78000000
Loading Purify'd Purify.rsc at 0x10000000
Starting main
Loading Purify'd MFC70ENU.DLL at 0x5d360000
HAN: Handle 0x00000154 is invalid in DeviceIoControl {1 occurrence}
HAN: Handle 0x00000154 is invalid in DeviceIoControl {7 occurrences}
Loading Purify'd COMRes.dll at 0x01840000 (preferred 0x77050000)
Loading Purify'd VERSION.dll at 0x67c00000
Loading Purify'd CLBCATQ.DLL at 0x76fd0000
Loading VERSION.dll at 0x77c00000
HAN: Handle 0x00000178 is invalid in RegNotifyChangeKeyValue {3 occurrences}
HAN: Handle 0x00000200 is invalid in DuplicateHandle {1 occurrence}
HAN: Handle 0x000001ed is invalid in WaitForSingleObjectEx {1 occurrence}
HAN: Handle 0x000001ed is invalid in WaitForSingleObjectEx {1 occurrence}
HAN: Handle 0x000001ed is invalid in WaitForSingleObjectEx {1 occurrence}
Starting thread 0xd90
Loading Purify'd COMCTL32.dll at 0x67340000
Loading COMCTL32.dll at 0x77340000
Loading Purify'd SHELL32.dll at 0x773d0000
Loading Purify'd COMCTL32.dll at 0x61950000
Loading COMCTL32.dll at 0x71950000
UMR: Uninitialized memory read in CWnd::ReflectChildNotify(UINT,UINT,long,long *) {2 occurrences}
Summary of all memory leaks... {35277 bytes, 20 blocks}
  MPK: Potential memory leak of 264 bytes from 1 block allocated in NdrCorrelationFree [RPCRT4.dll]
  MPK: Potential memory leak of 264 bytes from 1 block allocated in NdrCorrelationFree [RPCRT4.dll]
  MLK: Memory leak of 32 bytes from 1 block allocated in TranslateMDISysAccel [USER32.dll]
  MLK: Memory leak of 38 bytes from 1 block allocated in SetSystemMenu [USER32.dll]
  MPK: Potential memory leak of 1544 bytes from 1 block allocated in DPA_EnumCallback [COMCTL32.dll]
  MPK: Potential memory leak of 20 bytes from 1 block allocated in <>=(UINT) [MFC70D.DLL]
  MPK: Potential memory leak of 140 bytes from 5 blocks allocated in CAfxStringMgr::Allocate(int,int) [MFC70D.DLL]
  MPK: Potential memory leak of 113 bytes from 1 block allocated in CAfxStringMgr::Allocate(int,int) [MFC70D.DLL]
  MPK: Potential memory leak of 61 bytes from 3 blocks allocated in CAfxStringMgr::Allocate(int,int) [MFC70D.DLL]
  MPK: Potential memory leak of 33 bytes from 1 block allocated in std::basic_string<char,char_traits<char>::std,allocator<char>::std>:
  MLK: Memory leak of 12288 bytes from 1 block allocated in CAtlAllocator::AddModule(HINSTANCE__ *) [MFC70D.DLL]
  MLK: Memory leak of 8192 bytes from 1 block allocated in CAtlAllocator::AddCategory(int,WORD const*) [Scribble.exe]
  MLK: Memory leak of 8192 bytes from 1 block allocated in CAtlAllocator::AddCategory(int,WORD const*) [MFC70D.DLL]
  MLK: Memory leak of 4096 bytes from 1 block allocated in CAtlAllocator::AddCategory(int,WORD const*) [MFC70D.DLL]
```

Displayed Errors: 0 of 0 Displayed Warnings: 23 of 23 Bytes leaked: 32838+2439

Ready

Purify Shortcomings

- Purify does not detect corruption on the **stack** or in **static memory** at all, only for dynamic memory.

Electric Fence (Linux)

- malloc debugger
- Allocate arrays in separate virtual memory segments.
- Use memory management *hardware* checking to detect out-of-bounds references.

from 'man efence'

Electric Fence helps you detect two common programming bugs: software that overruns the boundaries of a malloc() memory allocation, and software that touches a memory allocation that has been released by free(). Unlike other malloc() debuggers, Electric Fence will detect read accesses as well as writes, and it will pinpoint the exact instruction that causes an error. It has been in use at Pixar since 1987, and at many other sites for years.

Electric Fence uses the virtual memory hardware of your computer to place an inaccessible memory page immediately after (or before, at the user's option) each memory allocation. When software reads or writes this inaccessible page, the hardware issues a segmentation fault, stopping the program at the offending instruction. It is then trivial to find the erroneous statement using your favorite debugger. In a similar manner, memory that has been released by free() is made inaccessible, and any code that touches it will get a segmentation fault.

Simply linking your application with libefence.a will allow you to detect most, but not all, malloc buffer overruns and accesses of free memory. If you want to be reasonably sure that you've found all bugs of this type, you'll have to read and understand the rest of this man page.

Insure++
(Parasoft)

- Software solution, source-level
- Checks all types of memory references including those to static (global), stack, and shared memory.
- Can find memory corruption and memory leaks as well as errors allocating and freeing dynamic memory.
- Automatically finds (some of) these kinds of errors:
  - string manipulation errors,
  - operations on uninitialized pointers,
  - operations on pointers to unrelated data blocks,
  - invalid pointer operations,
  - incompatible variable declarations,
  - mismatched variable types.

Insure++
(Parasoft)

- Detection of memory corruption on heap and stack
- Detection of uninitialized variables, pointers, and objects
- Detection of memory leaks and other memory allocation/free errors
- STL checking for proper usage of STL containers and related memory errors
- Compile-time checks for type- and size-related errors
- Runtime tracing of function calls
- GUI and command line interface
- Memory error checking in 3rd party static and dynamic libraries
- Direct interfaces with Visual Studio debugger

Insure++ Mutation Testing

- Leverages techniques from traditional Mutation Testing to uncover ambiguities that are difficult to detect through other methods or tools.
- Whereas traditional Mutation Testing attempts to create "faulty" mutants to create a more effective test suite, Insure++ creates and executes **what should be functionally equivalent mutants** of the source code under test.
- **When one of these mutants performs differently** than the original program, it **indicates that the code's functionality relies on implicit assumptions which may not always be satisfied during execution.**
- If a mutant causes the program to crash or encounter other serious problems, it's a sign that when the ***assumptions are not satisfied***, serious errors will occur at runtime.

Insure++ re. C++

- Because Mutation Testing can uncover a number of otherwise hard-to-find or esoteric errors, it is particularly important in C++, where there are so many opportunities to make errors.

- For example, Mutation Testing can detect errors such as:
  - lack of copy constructors
  - bad copy constructors
  - missing or incorrect constructors
  - wrong order of initialization of code
  - problems with operations of pointers
  - dependence on undefined behavior, such as order of evaluation

Valgrind
(http://valgrind.org/)

- **Valgrind Memcheck** detects memory-management problems, aimed primarily at C and C++ programs.
- When a program is run under Memcheck's supervision, all reads and writes of memory are checked, and calls to malloc/new/free/delete are intercepted.
- As a result, Memcheck can detect:
  - Use of uninitialized memory
  - Reading/writing memory after it has been free'd
  - Reading/writing off the end of malloc'd blocks
  - Reading/writing inappropriate areas on the stack
  - Memory leaks -- where pointers to malloc'd blocks are lost forever
  - Passing of uninitialized and/or unaddressable memory to system calls
  - Mismatched use of malloc/new/new [] vs free/delete/delete []
  - Overlapping src and dst pointers in memcpy() and related functions
  - Some misuses of the POSIX pthreads API

Helgrind: Race Detector

- Now part of Valgrind suite.
- Identifies **locations that are accessed by more than one thread**.
- For each such location, records which of the program's (pthread_mutex_) locks were held by the accessing thread at the time of the access.
- Checks whether there is at least one lock which is used by all threads to protect that location.
- **If no such lock can be found**, then there is (apparently) no consistent locking strategy being applied for that location, and so a possible data race might result.
- Helgrind also allows for "thread segment lifetimes". If the execution of two threads cannot overlap -- for example, if your main thread waits on another thread with a pthread_join() operation -- they can both access the same variable without holding a lock.

Time Rover / State Rover

http://www.time-rover.com/

Real-time monitoring based on temporal-logic assertions in **UML State Charts**

Examples of Temporal Logic Assertions

- ev1 and ev2 happen or do not happen simultaneously:
  Always ({ ev1 } Iff { ev2 })

- if ev1 then ev2 two cycles later:
  Always ({ ev1 } Implies (2){ ev2 })

- ev2 not before ev1:
  Always (Not{ev2} Until {ev1})

- ev2 any number of cycles after ev1:
  Always ({ev1} Implies Eventually {ev2})

Static Tools

- Polyspace (Patrick Cousot, now part of The MathWorks)
- Coverity Prevent (Dawson Engler)
- Klocwork
- CodeSurfer (GrammaTech)
- LintPlus (Cleanspace)
- PREfix/PREfast
- SLAM
- ESP

Polyspace Errors Detected (C, C++, Ada)

- Non-initialized variable read attempt
- Access conflicts for unprotected shared data in multi threaded applications
- Referencing through null pointers
- Out-of-bounds array access
- Out-of-bounds pointers
- Illegal type conversion
- Invalid arithmetic operations
- Overflow / underflow Unreachable code
- Invalid dynamic casts calls
- Throws of unauthorized exceptions
- Negative size arrays
- Null receivers
- Null pointers to members
- Wrong type for receivers
- Throws during catch parameter construction

Polyspace Analysis Output

Green	The code is safe - no run-time error. *You can focus your efforts elsewhere.*
Red	A run-time error <u>will</u> occur every time the operation is executed. *A solution should be implemented.*
Grey	The code is unreachable. *May point to an unimplemented functionality of the program.*
Orange	Warning - a run-time error *may* happen depending on the circumstances encountered at run-time. *Often missed by other methods, these errors can nonetheless cause major issues.*

Procedural entities	!	✗	?	✓	Line	Col	Details
Demo_Cpp	2	10	24	219			
__polyspace__edgstubs					1		__polyspace__edgst...
canalogic	1	3	2	11	1		canalogic.cpp
CAnalogic::TypeInfo()	1	3	2	11	12	15	canalogic.cpp
✓ COR.0				1	12		function never raises...
✓ COR.1				1	12		function returns a val...
✓ COR.2				1	14	12	call never raises an e...
✓ COR.3				1	15	18	call never raises an e...
✓ ASRT.4				1	17	2	failure of user asserti...
✓ ASRT.5				1	18	2	failure of user asserti...
✓ COR.6				1	20	10	call never raises an e...
✓ COR.7				1	23	18	call never raises an e...
✓ COR.8				1	23	18	non null receiver
✗ COR.10		1			24	4	correct type for recei...
✗ COR.9		1			24	4	correct type for recei...
✗ IDP.11		1			24	29	pointer within bounds
! COR.13	1				24	29	correct typeid argum...
✓ COR.16				1	26	18	call never raises an e...
✓ COR.17				1	26	18	non null receiver
? COR.18			1		29	2	correct type for recei...
? COR.19			1		29	2	correct type for recei...

Polyspace Source Highlighting

```
1
2     #include <cassert>
3     #include <typeinfo>
4
5     #include "canalogic.h"
6     #include "zz_utils.h"
7
8
9     static Utils  u;
10
11
12    int CAnalogic::TypeInfo()
13    {
14      CAnalogic canal;
15      const CAnalogic logic;
16
17      assert(typeid(logic) == typeid(canal));
18      assert(typeid(Capteur) == typeid(const Capteur));
19
20      Capteur cap;
21      Capteur* ptr = 0;
22
23      if (u.random_int())
24        return (typeid(this) ==  typeid (*ptr));
25
26      if (u.random_int())
27        ptr = &cap;
28
29      return 0;
30    }
31
32
33    // end
```

Polyspace Demo (Movie)

http://www.mathworks.com/products/demos/polyspaceclientc/polyspace_intro/index.html

Coverity

- Seeded by research from Stanford professor Dawson Engler, et al.
- Target: "Commercial code typically has anywhere from **one to seven bugs per 1,000 LOC**, according to an April report from the National Cybersecurity Partnership's Working Group on the Software Lifecycle, which cited an analysis of development methods by the Software Engineering Institute at Carnegie Mellon University."

Coverity analyses

- MySQL: found an average of one bug in every 4,000 LOC --results that are at least four times better than is typical with commercial software.
- Linux kernel: found average of less than one flaw in every 10,000 LOC.

Coverity Prevent Movie

http://www.coverity.com/html/video_player.html?file=CIC-Screencast-2009-04.swf

Coverity Prevent: CID 676: USE_AFTER_FREE in handle_write_stream - Mozilla Firefox

File Edit View History Bookmarks Tools Help

http://localhost:5440/cov.cgi?events=4970&line=1898&prec=%2Fcov.cgi%3Fc%3DAAAAAADA7g%26

Most Visited | Getting Started | Latest Headlines | SalesForce.com | WikiCoverity | localhost:5467 | localhost:4000 | localhost:8090 | localhost:5440

Coverity Prevent: CID 676: USE_... | Architecture Analyzer Web App | Over...

Home | Defects | Charts | Admin — coverity

Products | Runs | Components | Files | Functions | Individual | Jump to CID: | Quick Links: | Help | Logout

Comment:

Class: Bug
Severity: Major
Action: Undecided
Owner: carmelo (admin, carmelo)
Ext. Ref.:

Update | Export

Comment History:

admin 2009-04-10 11:50
Classification: **Bug**
Severity: **Major**
Owner: **carmelo**

admin 2009-04-07 17:37
Classification: Uninspected
Severity: Unknown
Action: **Undecided**

Individual > Defect

Checker: USE_AFTER_FREE
Function: ACE_POSIX_Asynch_Transmit_Handler::handle_write_stream(const ACE_Asynch_Write_Stream::Result &)
File: /nfs/ccaisip/SE_training_04072009/ACE_wrappers/ace/POSIX_Asynch_IO.cpp

CID: 676 (3 of 3) | Show events inline | Find event: [POSIX_Asynch_IO.cpp:1898] freed_arg | Show cross references

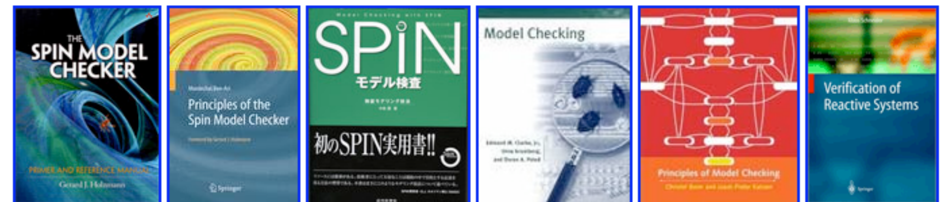
```
1898            delete this;
1899          }
1900      }
1901
1902    // Write stream successful.
1903
1904    // Partial write to socket.
1905    size_t unsent_data = result.bytes_to_write () - result.bytes_transferred ();
At conditional (1): "unsent_data != 0U" taking true path
1906    if (unsent_data != 0)
1907      {
1908        ACE_DEBUG ((LM_DEBUG,
At conditional (2): "false" taking false path
1909                    "%N:%l:Partial write to socket: Asynch_write called again\n"));
1910
1911        // Duplicate the message block and retry remaining data
Event deref_after_free: Dereferencing freed pointer "this"
1912        if (this->ws_.write (*result.message_block ().duplicate (),
1913                             unsent_data,
1914                             result.act (),
```

Model Checkers

- Generic model checking
  - Spin/Promela (http://spinroot.com/: Gerard Holzmann, Bell Labs, JPL)
  - SMV
  - Murphi
- Automatic model generation model checking
  - Verisoft
  - Pathfinder
  - Bandera
  - SLAM (sort of)
- UPPAAL (Uppsala and Aalborg Universities)
- SGM (State Graph Manipulation, Pao-Ann Hsiung)
- ASTRÉE

Spin

- Continuous development since 1980
- Used for space missions, many apps
- Many extensions built on it (open-source)
- Code expressed in Promela, a CSP-like concurrent programming language
- External calls to C possible



- Conditions expressed in LTL (Linear Temporal Logic)

Glimpses of Promela

do-statement (2)

- Example – modelling a traffic light

if- and do-statements are ordinary Promela statements; so they can be nested.

```
mtype = { RED, YELLOW, GREEN } ;
```

mtype (message type) models enumerations in Promela

```
active proctype TrafficLight() {
    byte state = GREEN;
    do
    ::  (state == GREEN)  -> state = YELLOW;
    ::  (state == YELLOW) -> state = RED;
    ::  (state == RED)    -> state = GREEN;
    od;
}
```

Note: this do-loop does not contain any non-deterministic choice.

Glimpses of Promela

Promela Example

```
byte x,t1,t2;

proctype Thread1()
{ do :: t1 = x;        1.1
        t2 = x;        1.2
        x = t1 + t2    1.3
  od }

proctype Thread2()
{ do :: t1 = x;        2.1
        t2 = x;        2.2
        x = t1 + t2    2.3
  od }

init
{ x = 1;               0.1
  run Thread1();       0.2
  run Thread2();       0.3
  assert(x != N) }     0.4
```

Another Violating schedule for $N = 2$

Step	State
(initial values)	[x = 0, t1 = 0, t2 = 0]
0.1	[x = 1, t1 = 0, t2 = 0]
0.2	[x = 1, t1 = 0, t2 = 0]
0.3	[x = 1, t1 = 0, t2 = 0]
2.1	[x = 1, t1 = 1, t2 = 0]
2.2	[x = 1, t1 = 1, t2 = 1]
2.3	[x = 2, t1 = 1, t2 = 1]
0.4	[x = 2, t1 = 1, t2 = 1]

violation

Uppaal

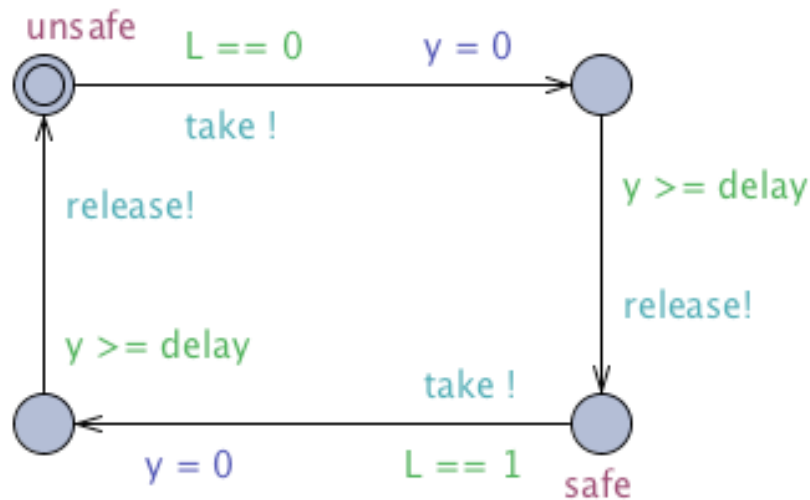
- From universities of Uppsala (Sweden) and Aalborg (Denmark)
- Based on intercommunicating ***timed*** **automata** (ta)
- Graphical input language, with textual counterpart

Glimpses of Uppaal
(bridge & flashlight puzzle)

```
/* Four vikings are about to cross a damaged bridge in the middle of the
 * night. The bridge can only carry two of the vikings at the time and to
 * find the way over the bridge the vikings need to bring a torch.  The
 * vikings need 5, 10, 20 and 25 minutes (one-way) respectively to cross
 * the bridge.

 * Does a schedule exist which gets all four vikings over the bridge
 * within 60 minutes?
 */

// Global declarations

chan take, release;         // Take and release torch
int[0,1] L;                 // The side the torch is on
clock time;                 // Global time
```

Glimpses of Uppaal

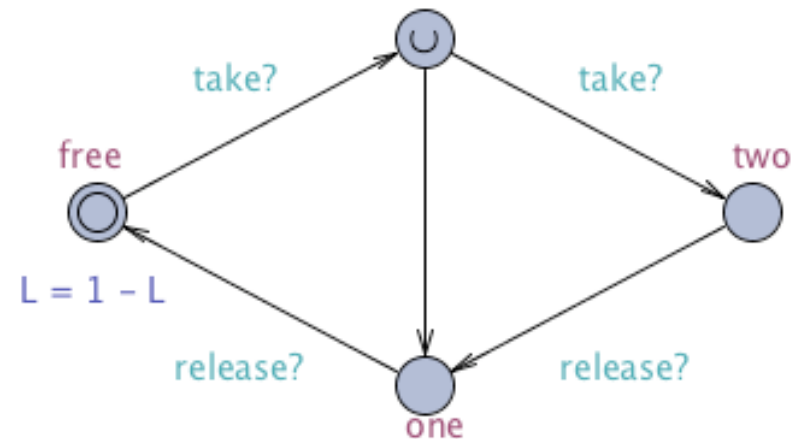
(bridge & flashlight puzzle)



Soldier timed automaton

```
// Soldier declarations

clock y;
```

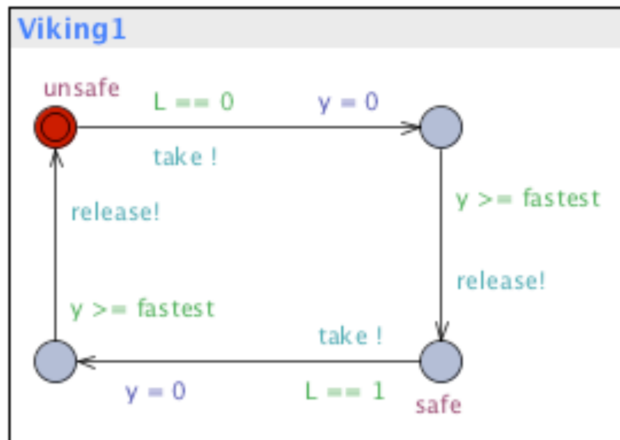
```
//System declarations

const int fastest = 5;
const int fast    = 10;
const int slow    = 20;
const int slowest = 25;

Viking1 = Soldier(fastest);
Viking2 = Soldier(fast);
Viking3 = Soldier(slow);
Viking4 = Soldier(slowest);

system Viking1, Viking2, Viking3, Viking4, Torch;
```

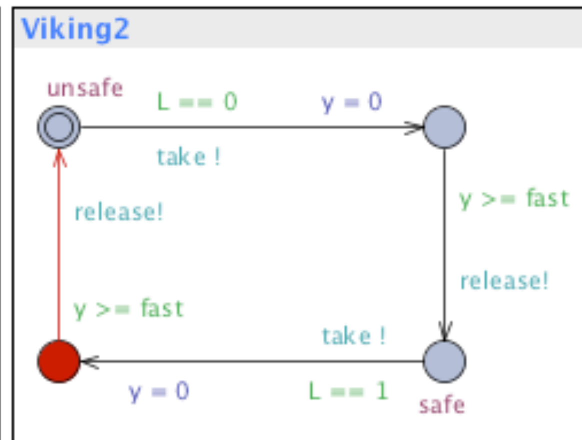


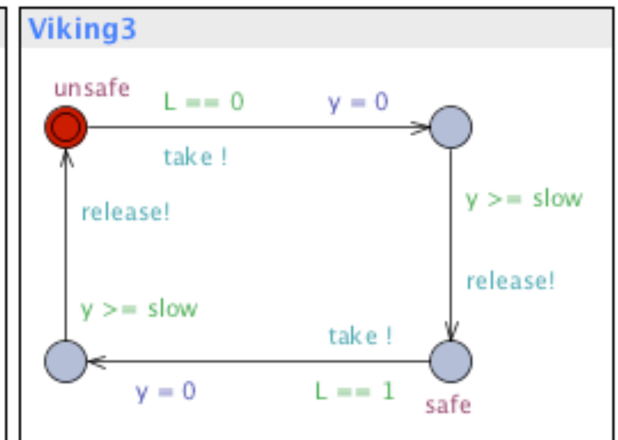
Torch timed automaton

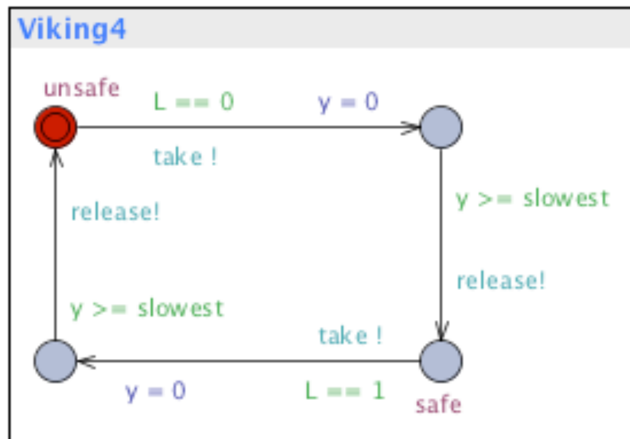
Glimpses of Uppaal

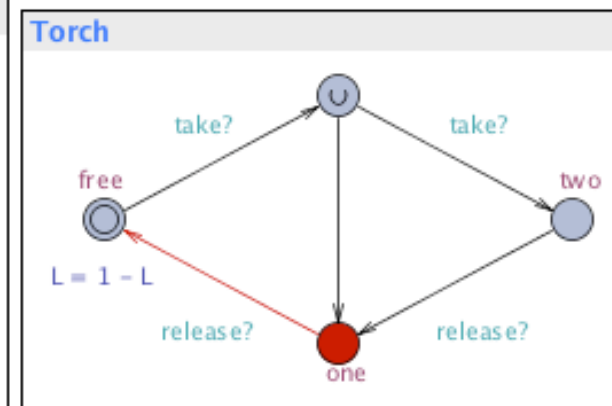
(bridge & flashlight puzzle)



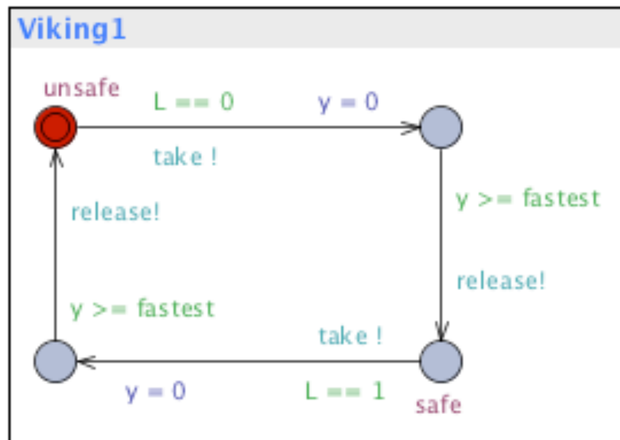


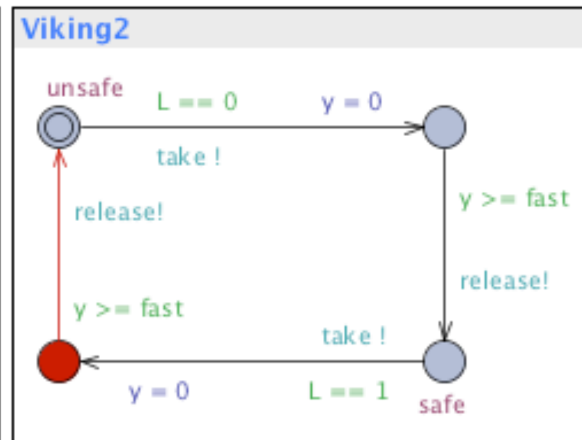


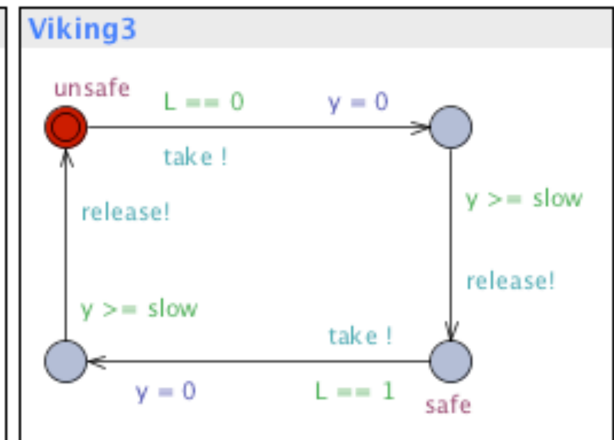


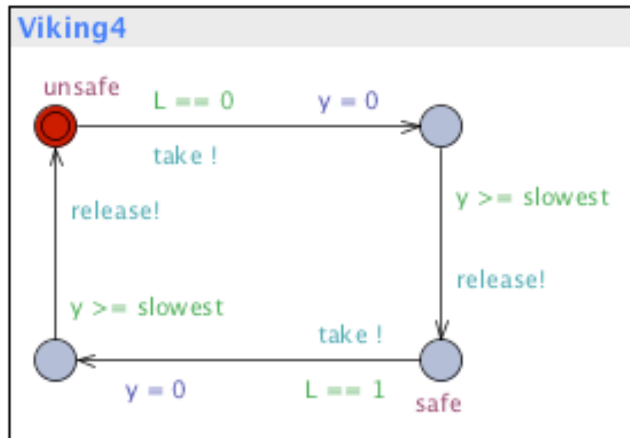


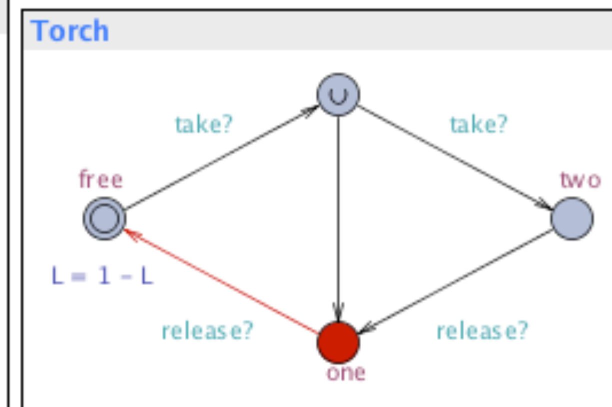
Glimpses of Uppaal
(bridge & flashlight puzzle)



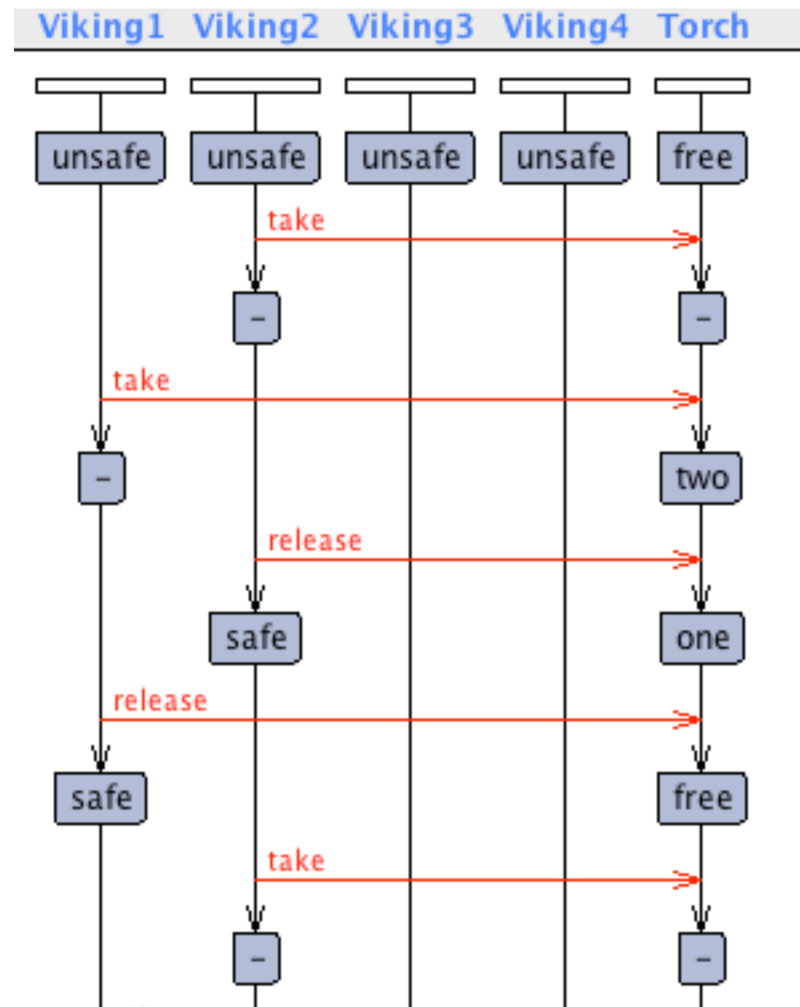








Auto-Generated Sequence Diagrams



Uppaal Assertions

```
A[] not deadlock
E<> Viking1.safe
E<> Viking2.safe
E<> Viking3.safe
A[] not (Viking4.safe and time<slowest)
E<> Viking4.safe imply time>=slowest
E<> Viking1.safe and Viking2.safe and Viking3.safe and Viking4.safe
```

Meaning of Uppaal Assertions

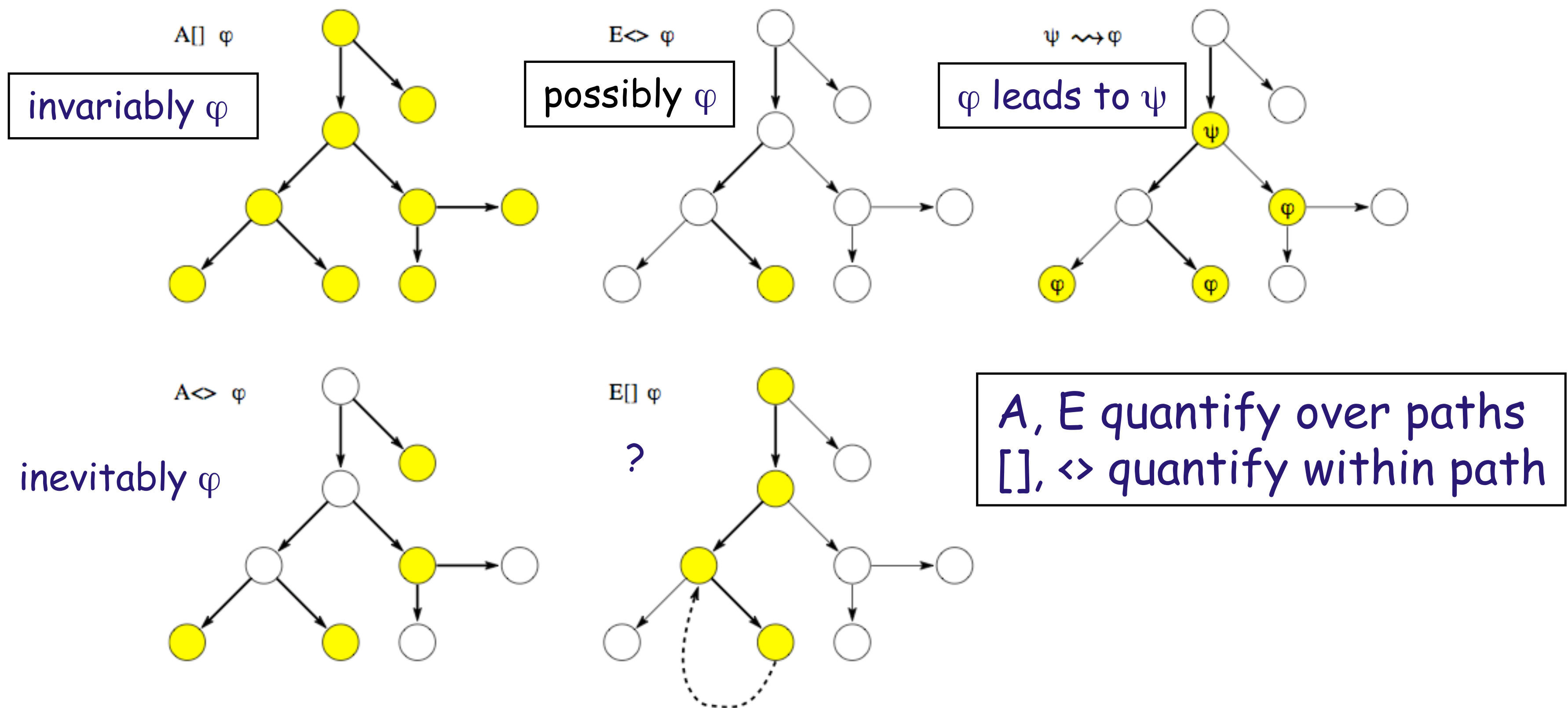
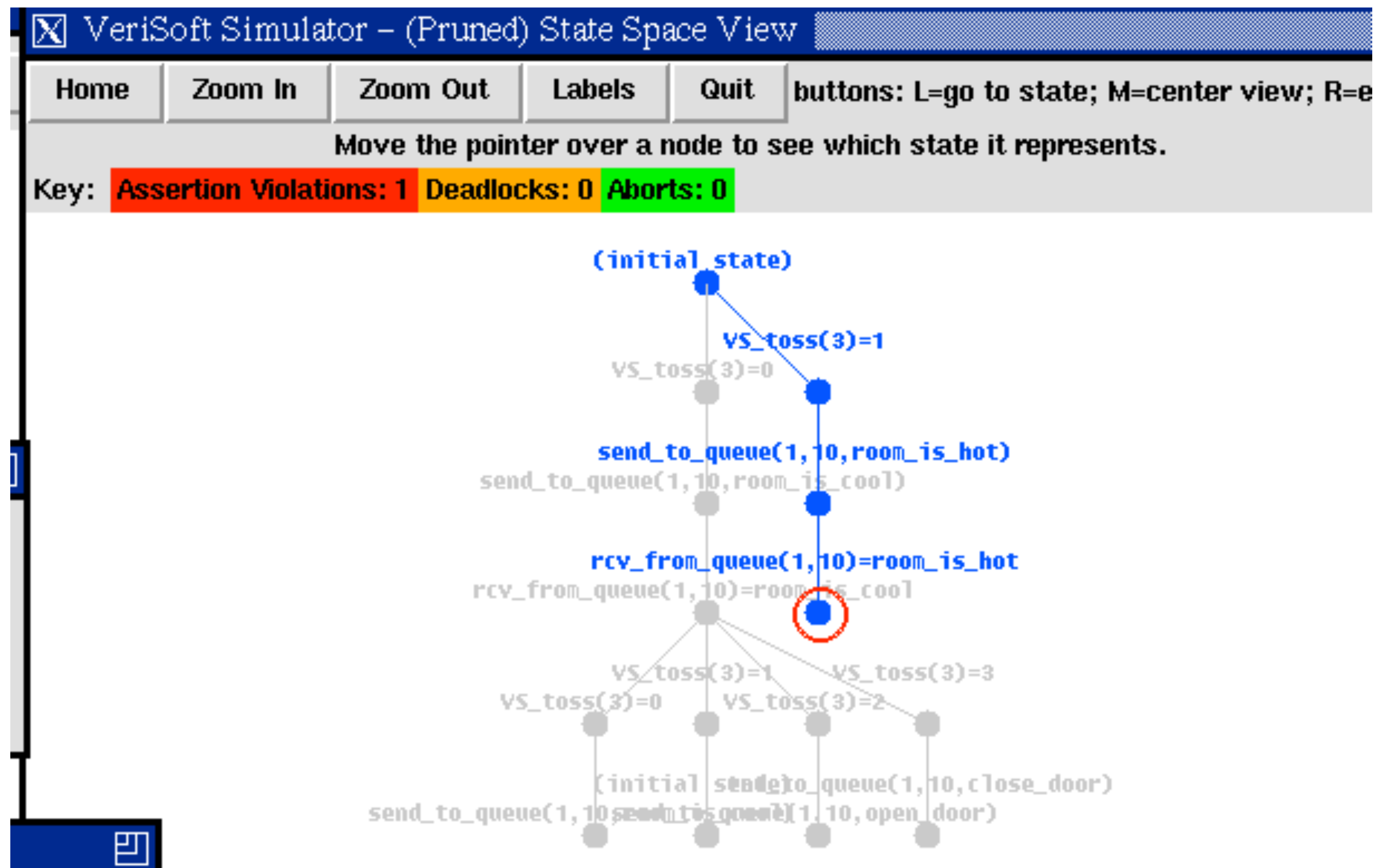


Fig. 3. Path formulae supported in UPPAAL. The filled states are those for which a given state formulae ϕ holds. Bold edges are used to show the paths the formulae evaluate on.

VeriSoft: Concurrent/Real-time



http://cm.bell-labs.com/who/god/verisoft/demo.html

VeriSoft Error Detection

- ***deadlocks***: a deadlock is a state of the state space where all processes are blocked.
- ***divergences***: a divergence occurs when a process does not attempt to communicate with the rest of the system for more than a given (user-specified) amount of time.
- ***livelocks***: a "livelock" occurs when a process is blocked during a sequence of more than a given (user-specified) number of successive states in the state space.
- ***assertion violations***: VS_assert operation can be inserted in the code of any process.

Excerpt from Case Study slides of Dawson Engler, founder of Coverity

We found bugs with static analysis and model checking and this is what we learned.

Dawson Engler and Madanlan Musuvathi

Based on work with

Andy Chou, David {Lie, Park, Dill}

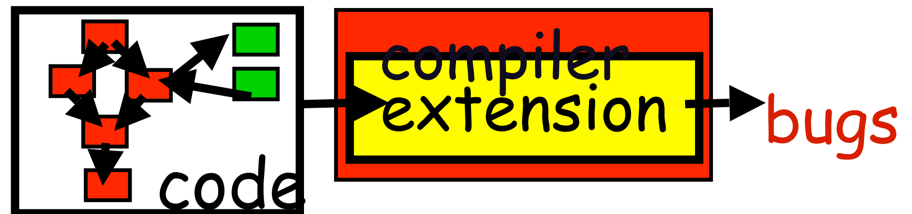
Stanford University

What's this all about

- A general goal of humanity: automatically find bugs
  - Success: lots of bugs, lots of code checked.
- Two promising approaches
  - Static analysis
  - Model checking
  - We used static analysis heavily for a few years & model checking for several projects over two years.
- General perception:
  - Static analysis: easy to apply but shallow bugs
  - Model checking: harder, but strictly better once done
  - Reality is a bit more subtle.

Crude definitions.

- "Static analysis" = our approach [DSL'97,OSDI'00]
  - Flow-sensitive, inter-procedural, extensible analysis
  - Goal: max bugs, min false pos
  - Not sound. No annotations.
  - Works well: 1000s of bugs in Linux, BSD, company code



- "Model checker" = explicit state space model checker
  - Use Murphi for FLASH, then home-grown for rest.
  - Limited domain: applying model checking to implementation code.

Cases

- Case 1: FLASH cache coherence protocol code
  - Checked w/ static analysis [ASPLOS'00]
  - Then w/ model checking [ISCA'01]
  - Surprise: static analysis found 4x more bugs.
- Case 2: AODV loop free, ad-hoc routing protocol
  - Checked w/ model checking [OSDI'02]
  - Took 3+ weeks; found ~ 1 bug / 300 lines of code
  - Checked w/ static (2 hours): more bugs when overlap
- Case 3: Linux TCP
  - Model checking: 6 months, 4 "ok" bugs.
  - Surprise: So hard to rip TCP out of Linux that it was easier to jam Linux into model checker

Some caveats

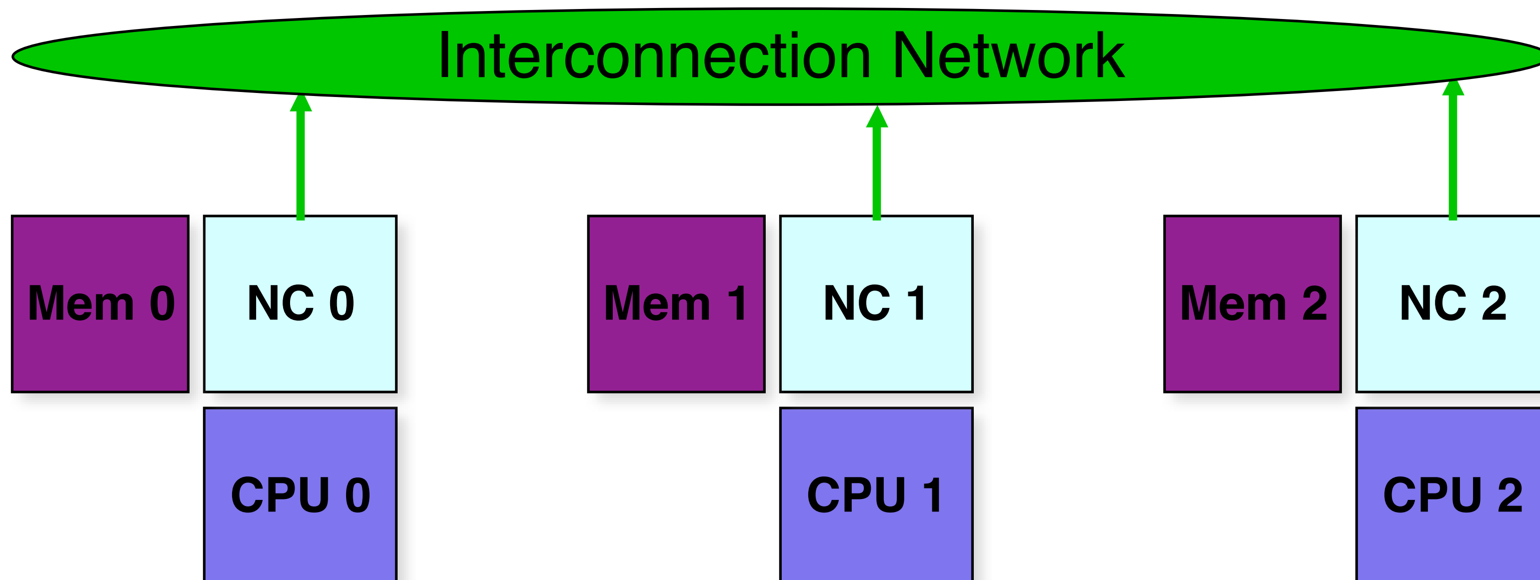
- Main bias:
  - Static analysis guy that happens to do model checking.
  - Some things that surprise me will be obvious to you.

  - We want model checking to succeed.
  - We're going to write a bunch more papers on it.
  - Life has just not always been exactly as expected.

- Of course
  - This is just a bunch of personal case studies
  - started up with engineers induction
  - to look like general principles. (1,2,3=QED)
  - While coefficients may change, general trends should hold

Case Study: FLASH

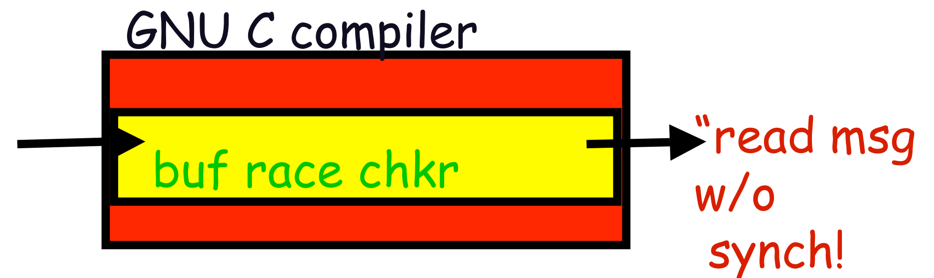
- ccNUMA with cache coherence protocols in software.
  - Has to be extremely fast
  - BUT: 1 bug deadlocks/livelocks entire machine
  - Heavily tested for 5 years.
  - Low-level with long code paths (73-183 LOC average)



Finding FLASH bugs with static analysis

- Gross code with many ad hoc correctness rules
  - But: they have a clear mapping to source code.
  - Easy to check with compiler.
- Example:
  - WAIT_FOR_DB_FULL must precede MISCBUS_READ_DB

```
Handler:
  if(...)
    WAIT_FOR_DB_FULL();

  MISCBUS_READ_DB();
```



  - Nice: scales, precise, statically found 34 bugs