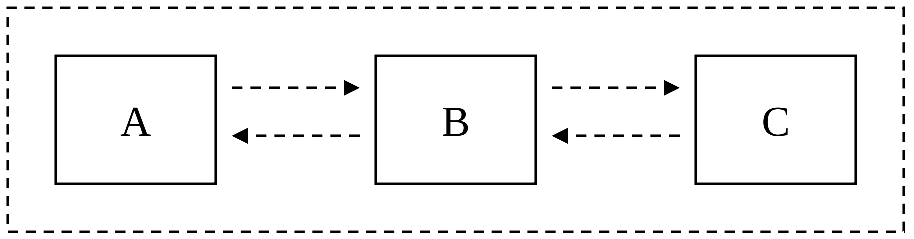
Between Testing and Verification:

Software Model Checking via Systematic testing

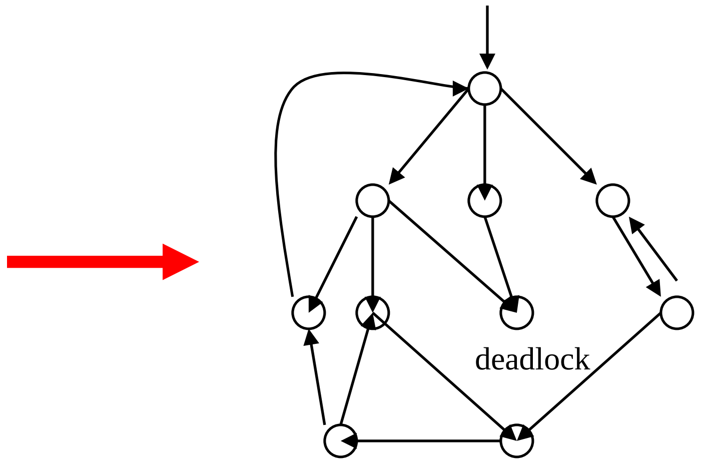
Patrice Godefroid

Microsoft Research

“Model Checking“



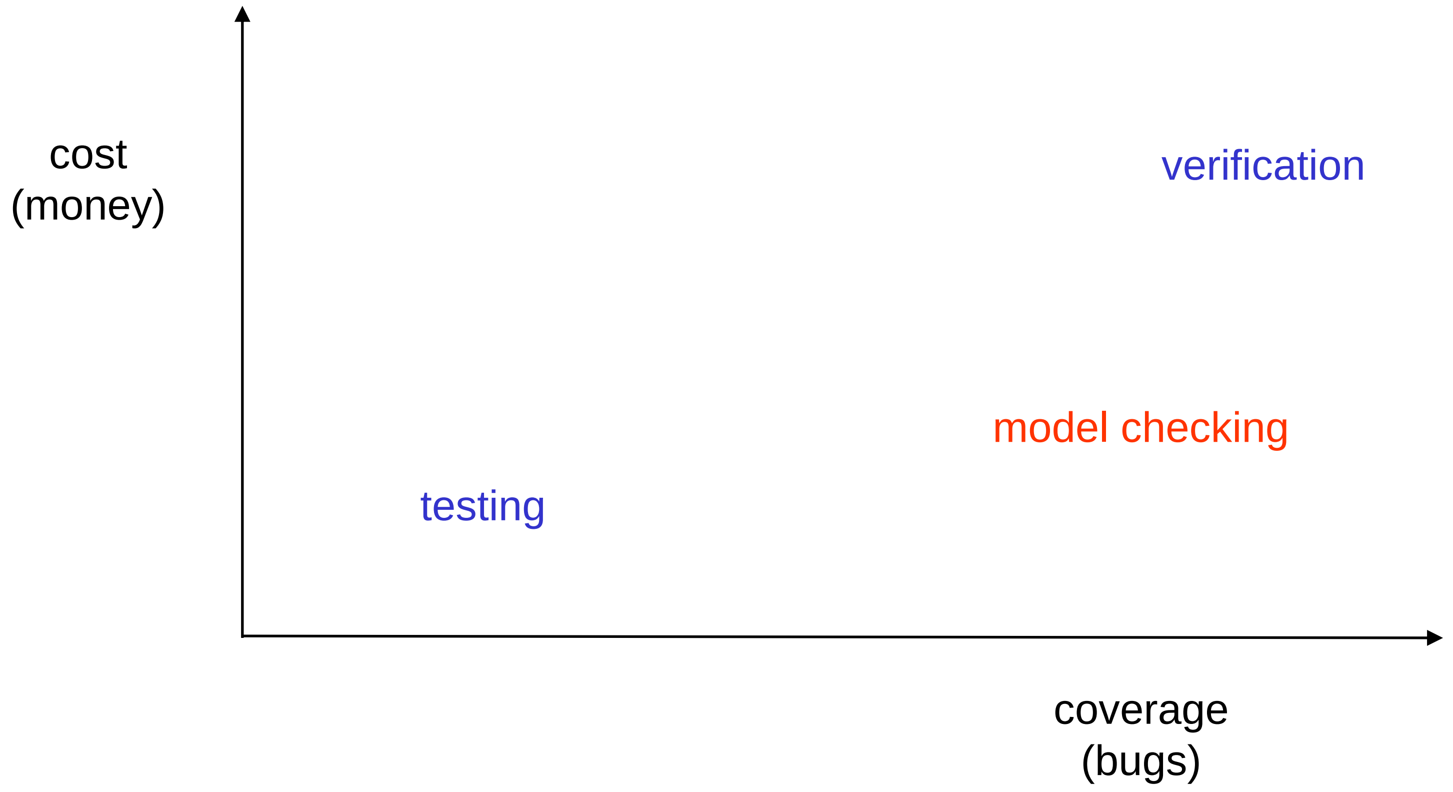
Each component is modeled by a FSM.



- Model Checking (MC) is
  - check whether a program satisfies a property by exploring its state space
  - systematic state-space exploration = exhaustive testing
  - “check whether the system satisfies a temporal-logic formula”
- Simple yet effective technique for finding bugs in high-level hardware and software designs
- Once thoroughly checked, models can be compiled and used as the core of the implementation

Insight: Model Checking is Super Testing

- Simple yet effective technique for finding bugs



Software Model Checking

- 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:

$$\begin{tikzcd}
\text{Modeling languages} \arrow[r, "\text{state-space exploration}"] & \text{Model checking} \arrow[d, "\text{adaptation}"] \\
\text{Programming languages} \arrow[u, "\text{abstraction}"] \arrow[r, "\text{state-space exploration}"] & \text{Systematic testing}
\end{tikzcd}$$

(SLAM, Bandera, FeaVer, BLAST,…)

Dynamic

Concurrency: **VeriSoft, JPF, CMC, CHESS, Cuzz,…**
Data inputs: **DART, EXE, SAGE, Pex,…**

Part 1

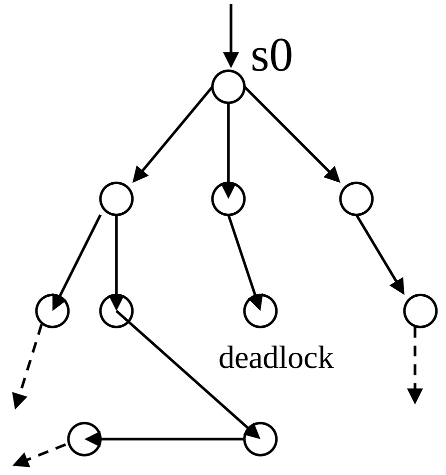
Dynamic Software Model Checking

Dealing with Concurrency

Dynamic Semantics (VeriSoft, POPL'97)

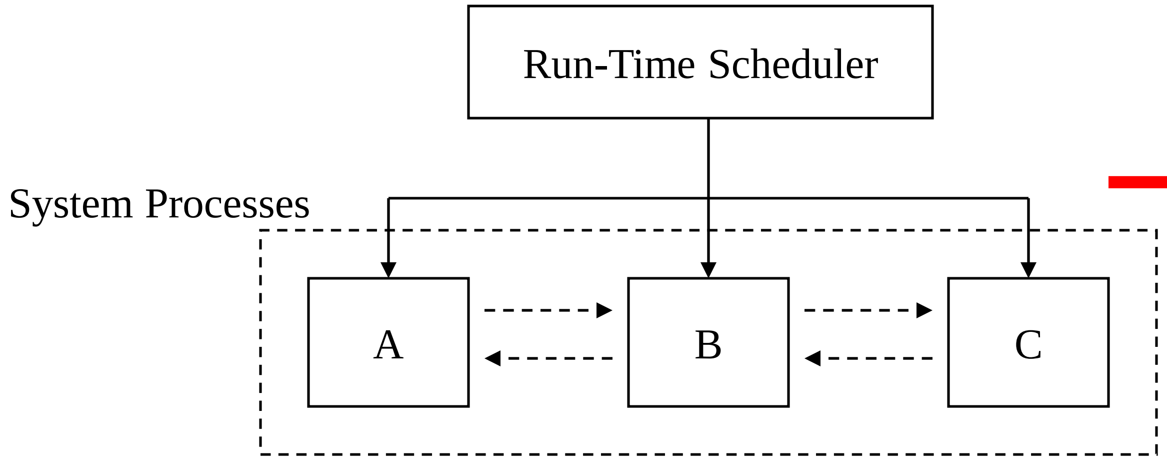
- State Space = "product of (OS) processes"
  - Processes communicate by executing <u>operations</u> on com. objects
  - Operations on com. objects are <u>visible</u>, other ops are <u>invisible</u>
  - Only executions of visible operations may be <u>blocking</u>
  - The system is in a <u>global state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

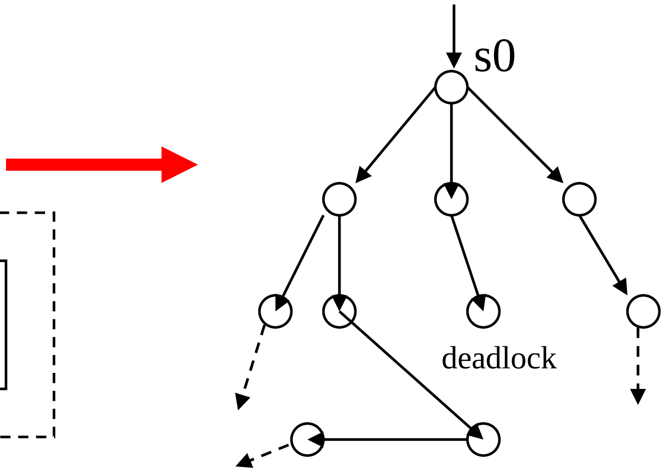
THEOREM: <u>Deadlocks</u> and <u>assertion violations</u> are preserved in the "state space" as defined above.



VeriSoft

- Controls and observes the execution of visible operations of concurrent processes by intercepting system calls (communication, assertion violations, etc.)
- Systematically drives the system along all the paths (=executions) in its state space (=automatically generate, execute and evaluate many executions)
- 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)"

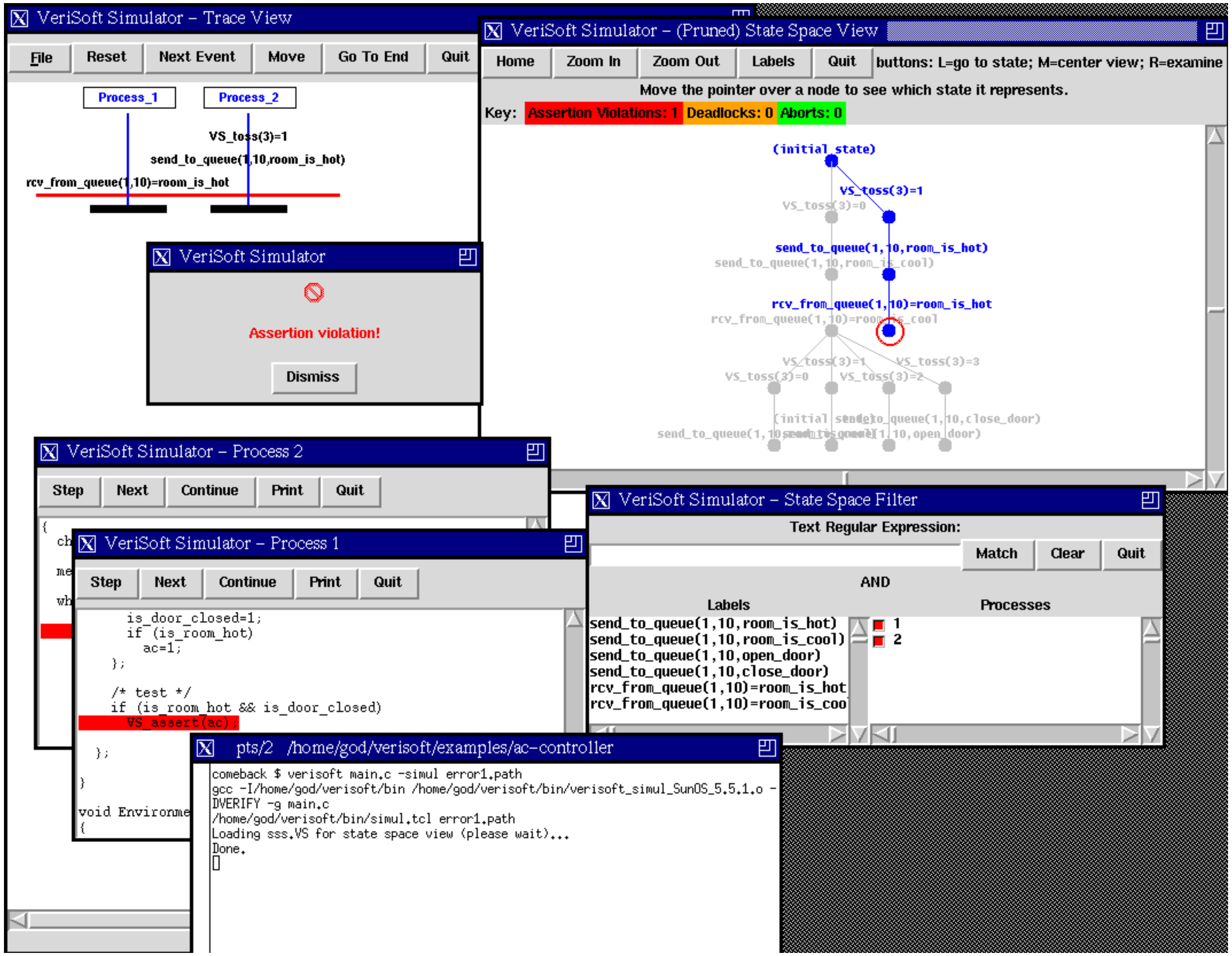




VeriSoft State-Space Search

- 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


VeriSoft Simulator – Trace View
File
Reset
Next Event
Move
Go To End
Quit
Process_1
Process_2
VS_toss(3)=1
send_to_queue(1,10,room_is_hot)
rcv_from_queue(1,10)=room_is_hot
VeriSoft Simulator – (Pruned) State Space View
Home
Zoom In
Zoom Out
Labels
Quit
buttons: L=go to state; M=center view; R=examine
Move the pointer over a node to see which state it represents.
Key: Assertion Violations: 1 Deadlocks: 0 Aborts: 0
(initial_state)
VS_toss(3)=1
VS_toss(3)=0
send_to_queue(1,10,room_is_hot)
send_to_queue(1,10,room_is_cool)
rcv_from_queue(1,10)=room_is_hot
rcv_from_queue(1,10)=room_is_cool
VS_toss(3)=1
VS_toss(3)=3
VS_toss(3)=0
VS_toss(3)=2
send_to_queue(1,10,close_door)
send_to_queue(1,10,open_door)
VeriSoft Simulator
Assertion violation!
Dismiss
VeriSoft Simulator – Process 2
Step
Next
Continue
Print
Quit
VeriSoft Simulator – Process 1
Step
Next
Continue
Print
Quit
VeriSoft Simulator – State Space Filter
Text Regular Expression:
Match
Clear
Quit
AND
Labels
Processes
1
2
pts/2 /home/god/verisoft/examples/ac-controller

```
      is_door_closed=1;
      if (is_room_hot)
        ac=1;
    };

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

  };

}
void Environme
{
```

```
send_to_queue(1,10,room_is_hot)
send_to_queue(1,10,room_is_cool)
send_to_queue(1,10,open_door)
send_to_queue(1,10,close_door)
rcv_from_queue(1,10)=room_is_hot
rcv_from_queue(1,10)=room_is_coo
```

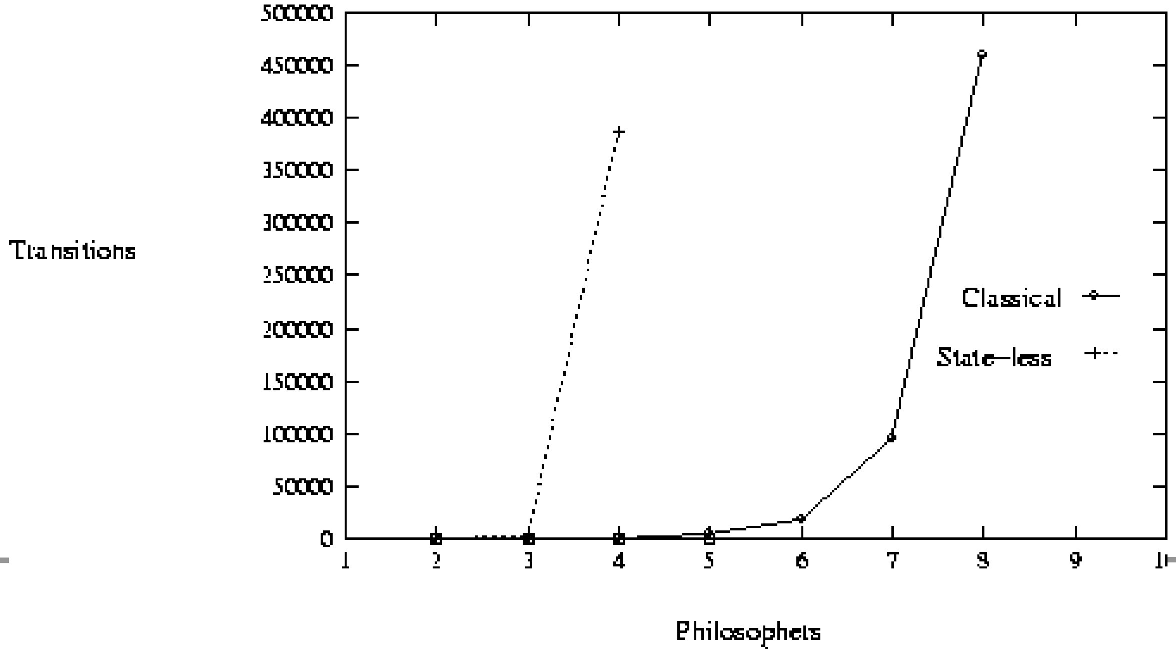
```
comeback $ verisoft main.c -simul error1.path
gcc -I/home/god/verisoft/bin /home/god/verisoft/bin/verisoft_simul_SunOS_5.5.1.o -
DVERIFY -g main.c
/home/god/verisoft/bin/simul.tcl error1.path
Loading sss.VS for state space view (please wait)...
Done.
```

Originality of VeriSoft

- VeriSoft = first systematic state-space exploration tool for concurrent systems composed of processes executing arbitrary code (e.g., C, 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!



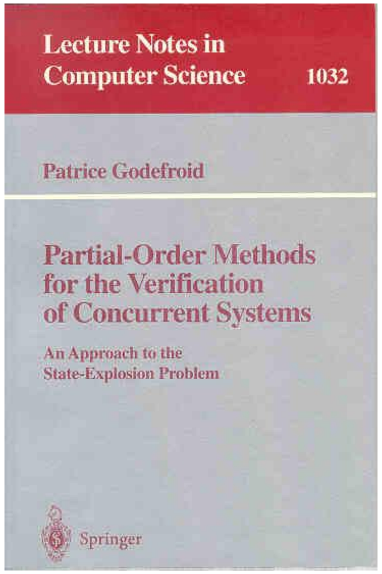
Partial-Order Reduction

- 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,...)

[Note: checking more elaborate properties require other extensions

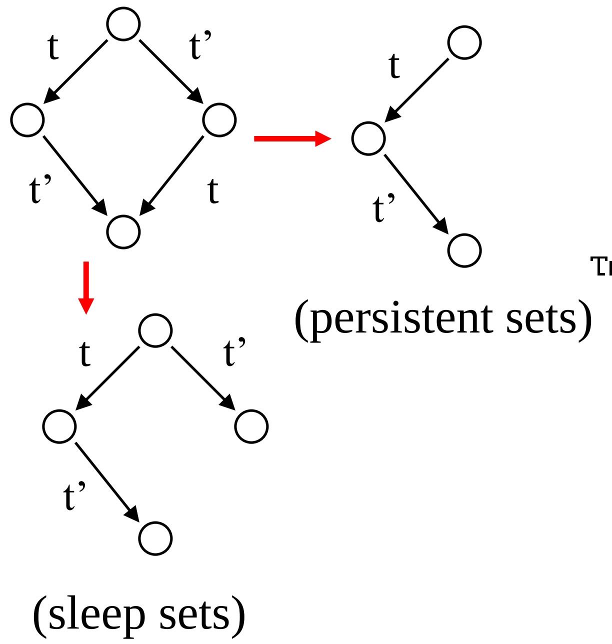
– Ex: ample sets (Peled) are persistent sets satisfying additional conditions sufficient for LTL model checking

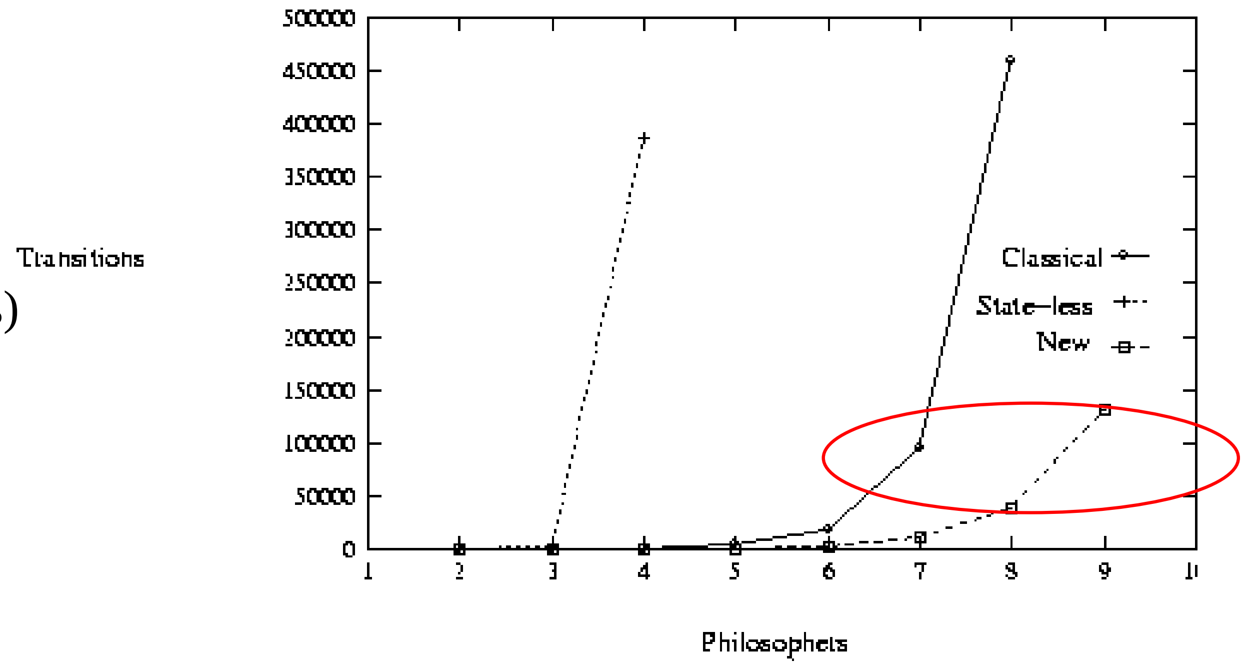
Not used here as VeriSoft only checks reachability properties]


Lecture Notes in Computer Science 1032
Patrice Godefroid
Partial-Order Methods for the Verification of Concurrent Systems
An Approach to the State-Explosion Problem
Springer

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 systematic state-less search in the state space of a concurrent system is untractable

VeriSoft - Summary

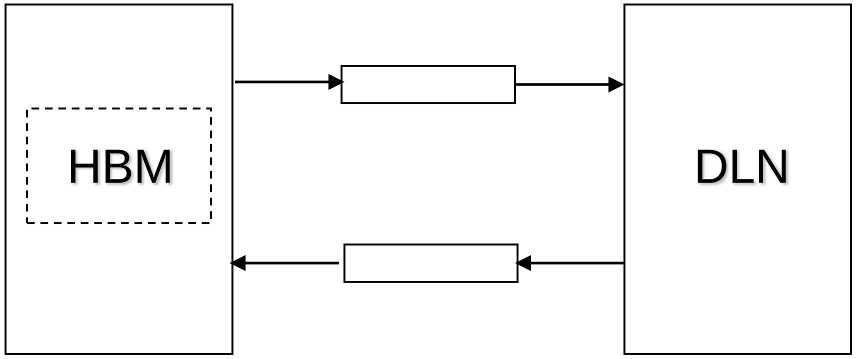
- Two key innovations:
  1. Does not use any specific modeling/programming language
  2. Performs a state-less search
- Use of partial-order reduction is key in presence of concurrency
- In practice, the search is typically incomplete !
- From a given initial state, can always guarantee a complete coverage 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)

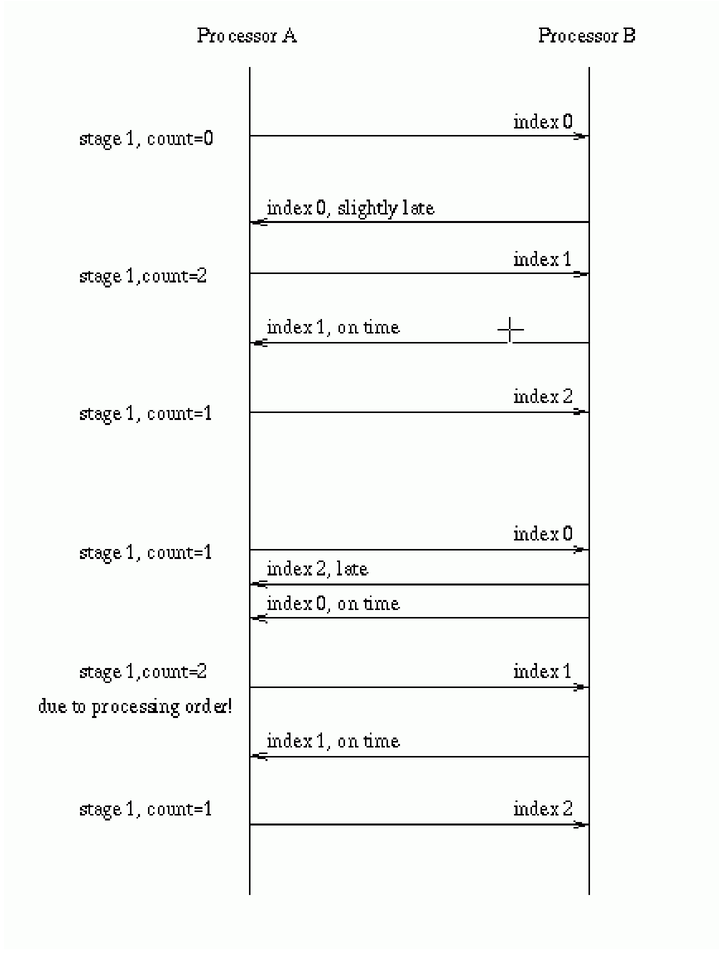
Application: 4ESS HBM (Small)

- HBM code = 100s lines of EPL (assembly) , controls millions of calls every day, behaves unexpectedly in the field
- 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)" (took only a few hours!)



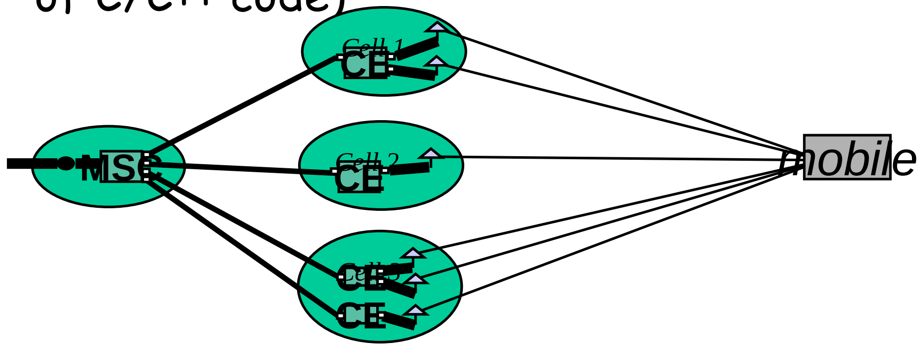
- Discovered several flaws in software and its documentation... [ISSTA98]

Example of scenario found:

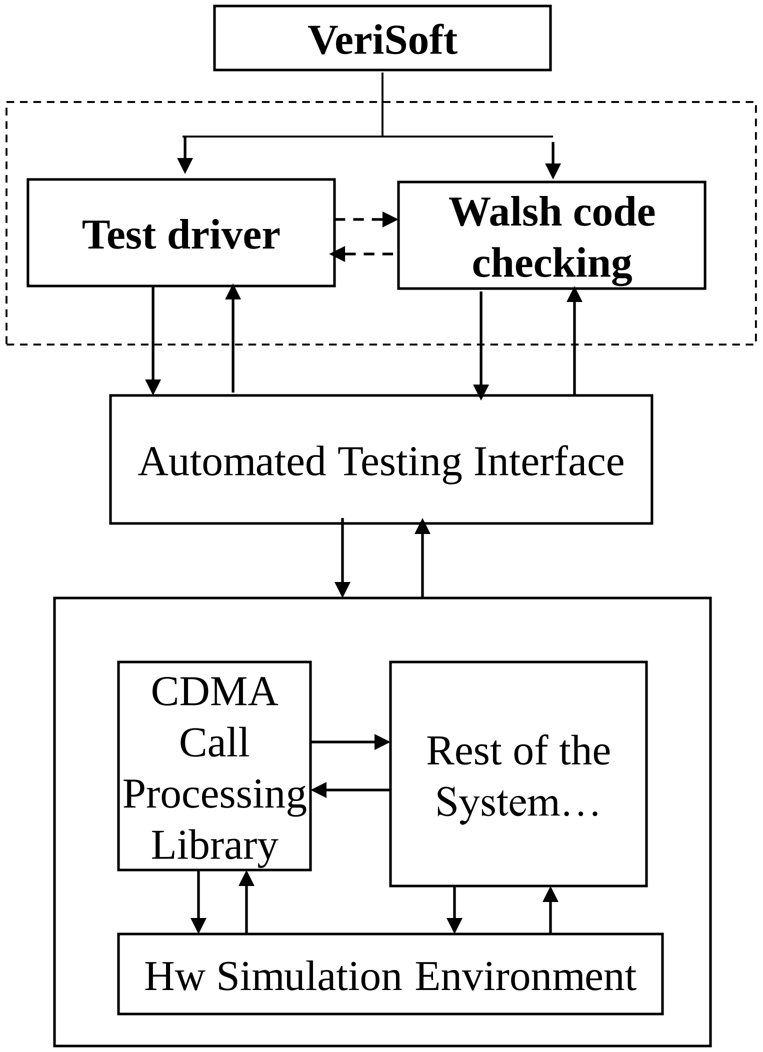


Application: CDMA Call Processing (Large)

- 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]



VeriSoft
Test driver
Walsh code checking
Automated Testing Interface
CDMA Call Processing Library
Rest of the System…
Hw Simulation Environment

Discussion

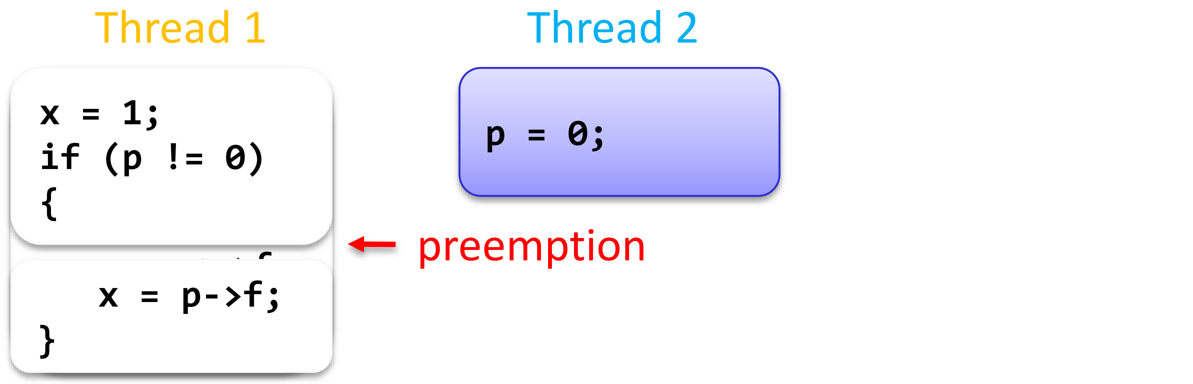
- VeriSoft (like model checking) is not a panacea
  - Limited by the state-explosion problem,…
  - Requires some training and effort (to write test drivers, properties…)
- Used properly, VeriSoft is very effective at finding bugs
  - Cheap, scalable (applicable to large systems) although incomplete
  - Concurrent/reactive systems are hard to design, develop and test
  - Traditional testing is not adequate
  - "Model checking" (systematic testing) can rather easily find new bugs
- These bugs would otherwise be found by the customer !
- So the real question is "How much ($) do you care about bugs?"

What about Multi-Threaded Programs?

- Software model checking via systematic testing works well for message-passing programs
  - Systematically exploring their state spaces up to (say) 50 message exchanges cover a lot of their functionality
- What about shared-memory programs?
  - Up to 50 read/write covers nothing !
  - Much more challenging…
- Some useful techniques:
  - Dynamic Partial Order Reduction [POPL'05] (10+ tools)
  - Preemption bounding (e.g., CHESS)
  - Concurrency heuristics (e.g., Cuzz)

CHESS (MSR): Preemption Bounding

- For multi-threaded concurrent software (Win32, CLR)
- Focus on executions with small number of preemptions
  - Heuristic: many bugs can be found with few preemptions



```
x = 1;
if (p != 0)
{
```

```
p = 0;
```

```
    x = p->f;
}
```

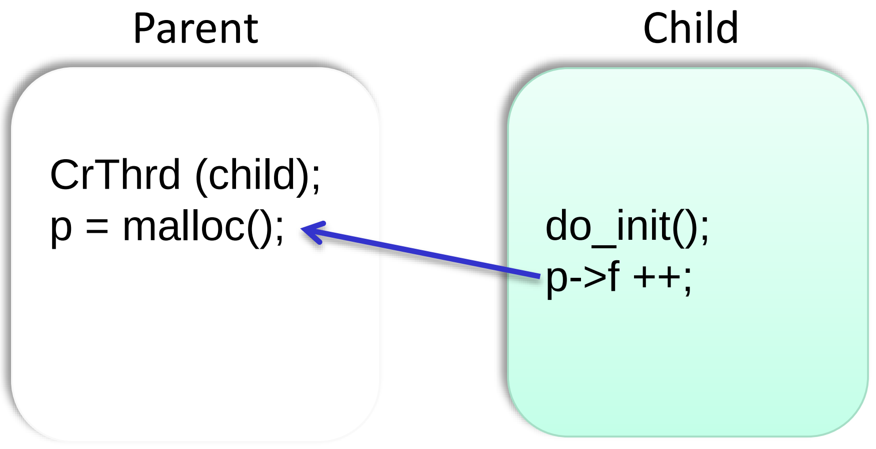
- Many bugs found this way
  - Can deal with very larger state spaces, complementary to (D)POR

CHESS Status

- Open source at: http://chesstool.codeplex.com/
- Platform for concurrency research [Musuvathi, Qadeer,…]
  - Preemption bounding [PLDI '07]
  - Fair stateless model checking [PLDI '08]
  - Weak memory model verification [CAV '08]
  - Data-race detection [PLDI '09]
  - Concurrency coverage [TACAS '08]
  - Linearizability checking [PLDI '10]
  - Partial-order reduction and preemption bounding [OOPSLA '13]
- Used to systematically test concurrency libraries in MS products
  - Task parallel library (TPL), Concurrency runtime (ConcRT), Concurrency coordination runtime (CCR), Singularity
  - Both to find unknown bugs and to reproduce known bugs

Other: Concurrency Heuristics

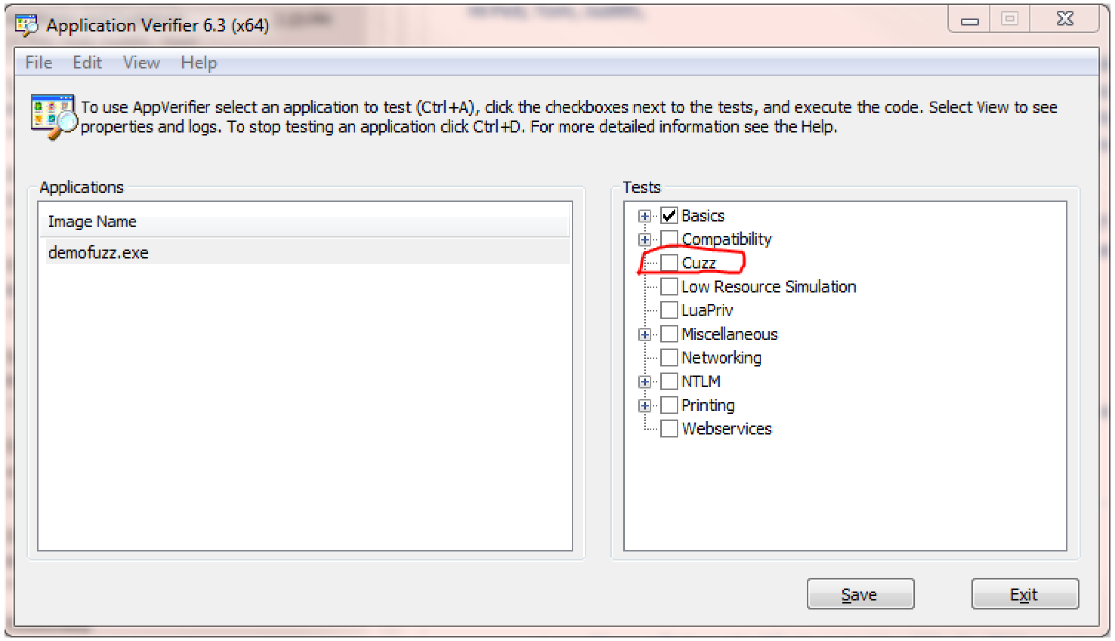
- Heuristics for partially exploring large state spaces
  - Genetic algorithms (with property-specific fitness functions)
  - Heuristics based on concurrent dependencies



If dereference before initialization, BUG!
Thus, ONE ordering constraint is sufficient for this bug
→ heuristic = delay malloc() as much as possible!

Cuzz (MSR): Concurrency Fuzzing

- Randomize thread schedules by delaying threads, with probability guarantees [ASPLOS'10]
- Now part of MS AppVerifier and Driver Verifier



Application Verifier 6.3 (x64)

To use AppVerifier select an application to test (Ctrl+A), click the checkboxes next to the tests, and execute the code. Select View to see properties and logs. To stop testing an application click Ctrl+D. For more detailed information see the Help.

Applications

Image Name
demofuzz.exe

Tests

- [✓] Basics
- [ ] Compatibility
- [ ] Cuzz
- [ ] Low Resource Simulation
- [ ] LuaPriv
- [ ] Miscellaneous
- [ ] Networking
- [ ] NTLM
- [ ] Printing
- [ ] Webservices

- Found bugs in large MS products (SQL server, IE,…)
  - Increase the coverage of existing tests
  - Increase the reproducibility of bugs

Some Other Related Tools

- Java PathFinder : uses a modified Java Virtual Machine
- CMC : stores partial state representations
- MaceMC : for Mace DSL, heuristics for 'liveness' properties
- MoDist : for distributed systems
- ISP : for MPI programs
- Etc. (the list above is not exhaustive !)

Part 2

Dynamic Software Model Checking

Dealing with Data Inputs

Automatic Code-Driven Test Generation

Problem:

Given a sequential program with a set of input parameters, generate a set of inputs that maximizes code coverage

= "automate test generation using program analysis"

Example: `Powerpnt.exe <filename>`

- Millions of lines of C/C++, complex input format, dynamic memory allocation, data structures of various shapes and sizes, pointers, loops, procedures, libraries, system calls, etc.

How? (1) Static Test Generation

- Static analysis to partition the program's input space [King76,…]

- Ineffective whenever symbolic reasoning is not possible
  - which is frequent in practice… (pointer manipulations, complex arithmetic, calls to complex OS or library functions, etc.)

Example:

```
int obscure(int x, int y) {
  if (x==hash(y)) error();
  return 0;
}
```

Can't statically generate values for x and y that satisfy "x==hash(y)" !

How? (2) Dynamic Test Generation

- Run the program (starting with some random inputs), gather constraints on inputs at conditional statements, use a constraint solver to generate new test inputs
- Repeat until a specific program statement is reached [Korel90,…]
- Or blend with model checking !
  - repeat to try to cover ALL feasible program paths
  - DART = Directed Automated Random Testing = systematic dynamic test generation [PLDI'05,…]
  - detect crashes, assertion violations, use runtime checkers (Purify, Valgrind, AppVerifier,…)

DART = Directed Automated Random Testing

Example:

```
int obscure(int x, int y) {
  if (x==hash(y)) error();
  return 0;
}
```

Run 1 : - start with (random) x=33, y=42
- execute concretely and symbolically:
  if (33 != 567) | if $\boxed{\text{(x != hash(y))}}$
  constraint too complex
  → simplify it: x != 567
- solve: x==567 → solution: x=567
- new test input: x=567, y=42

Run 2 : the other branch is executed
All program paths are now covered !

- Observations:
  - Dynamic test generation extends static test generation with additional runtime information: it is more powerful
    - see [DART in PLDI'05], [PLDI'11]
  - The number of program paths can be infinite: may not terminate!
  - Still, DART works well for small programs (1,000s LOC)
  - Significantly improves code coverage vs. random testing

DART Implementations

- Defined by symbolic execution, constraint generation and solving
  - Languages: C, Java, x86, .NET,…
  - Theories: linear arithmetic, bit-vectors, arrays, uninterpreted functions,…
  - Solvers: lp_solve, CVCLite, STP, Disolver, Z3,…
- Examples of tools/systems implementing DART:
  - EXE/EGT (Stanford): independent ['05-'06] closely related work (became KLEE)
  - CUTE = same as first DART implementation done at Bell Labs
  - SAGE (MSR) for x86 binaries and merges it with "fuzz" testing for finding security bugs (more next)
  - PEX (MSR) for .NET binaries in conjunction with "parameterized-unit tests" for unit testing of .NET programs
  - YOGI (MSR) for checking the feasibility of program paths generated statically using a SLAM-like tool
  - Vigilante (MSR) for generating worm filters
  - BitScope (CMU/Berkeley) for malware analysis
  - CatchConv (Berkeley) focus on integer overflows
  - Splat (UCLA) focus on fast detection of buffer overflows
  - Apollo (MIT/IBM) for testing web applications

…and many more!

An Application: SAGE @ Microsoft

- #1 application of SMT solvers today (CPU usage)
- Why? Security Testing
- Software security bugs can be very expensive:
  - Cost of each Microsoft Security Bulletin: $Millions
  - Cost due to worms (Slammer, CodeRed, Blaster, etc.): $Billions
- Many security vulnerabilities are in file & packet parsers
  - Ex: MS Windows includes parsers for hundreds of file formats
- Security testing: "hunting for million-dollar bugs"
  - Write A/V (always exploitable), Read A/V (sometimes exploitable), NULL-pointer dereference, division-by-zero (harder to exploit but still DOS attacks), etc.

Hunting for Security Bugs

- Main techniques used by "black hats":
  - Code inspection (of binaries) and
  - Blackbox fuzz testing
- Blackbox fuzz testing:
  - A form of blackbox random testing [Miller+90]
  - Randomly fuzz (=modify) a well-formed input
  - Grammar-based fuzzing: rules that encode "well-formed"ness + heuristics about how to fuzz (e.g., using probabilistic weights)
- **Heavily** used in security testing
  - Simple yet effective: many bugs found this way…
  - At Microsoft, fuzzing is mandated by the SDL →

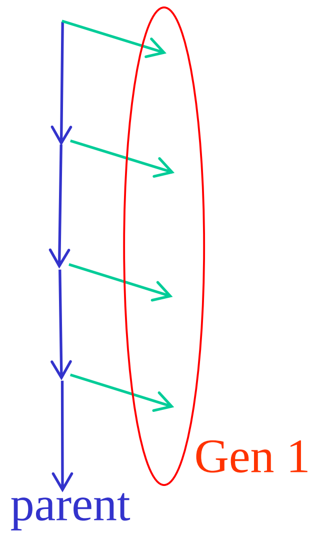
Introducing Whitebox Fuzzing [NDSS'08]

Idea: mix fuzz testing with dynamic test generation

- Dynamic symbolic execution to collect constraints on inputs, negate those, solve new constraints to get new tests, repeat → "systematic dynamic test generation" (= DART)

(Why dynamic ? Because most precise ! [PLDI'05, PLDI'11])

- Apply to large applications (not unit)
- Start with a well-formed input (not random)
- Combine with a generational search (not DFS)
  - Negate 1-by-1 each constraint in a path constraint
  - Generate many children for each parent run
  - Challenge all the layers of the application sooner
  - Leverage expensive symbolic execution
- Implemented in the tool SAGE

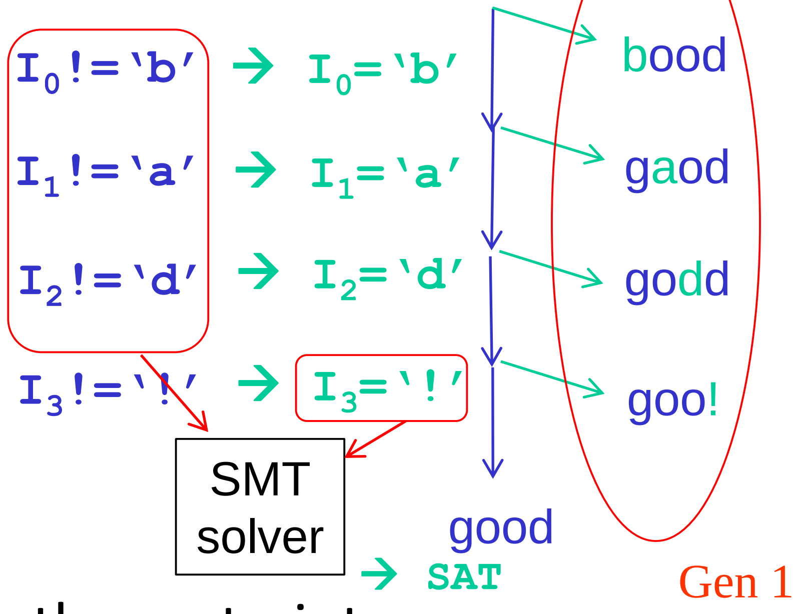


Example

```
void top(char input[4])
{
   int cnt = 0;
   if (input[0] == ‘b’) cnt++;
   if (input[1] == ‘a’) cnt++;
   if (input[2] == ‘d’) cnt++;
   if (input[3] == ‘!’) cnt++;
   if (cnt >= 4) crash();
}
```

input = “good”

Path constraint:



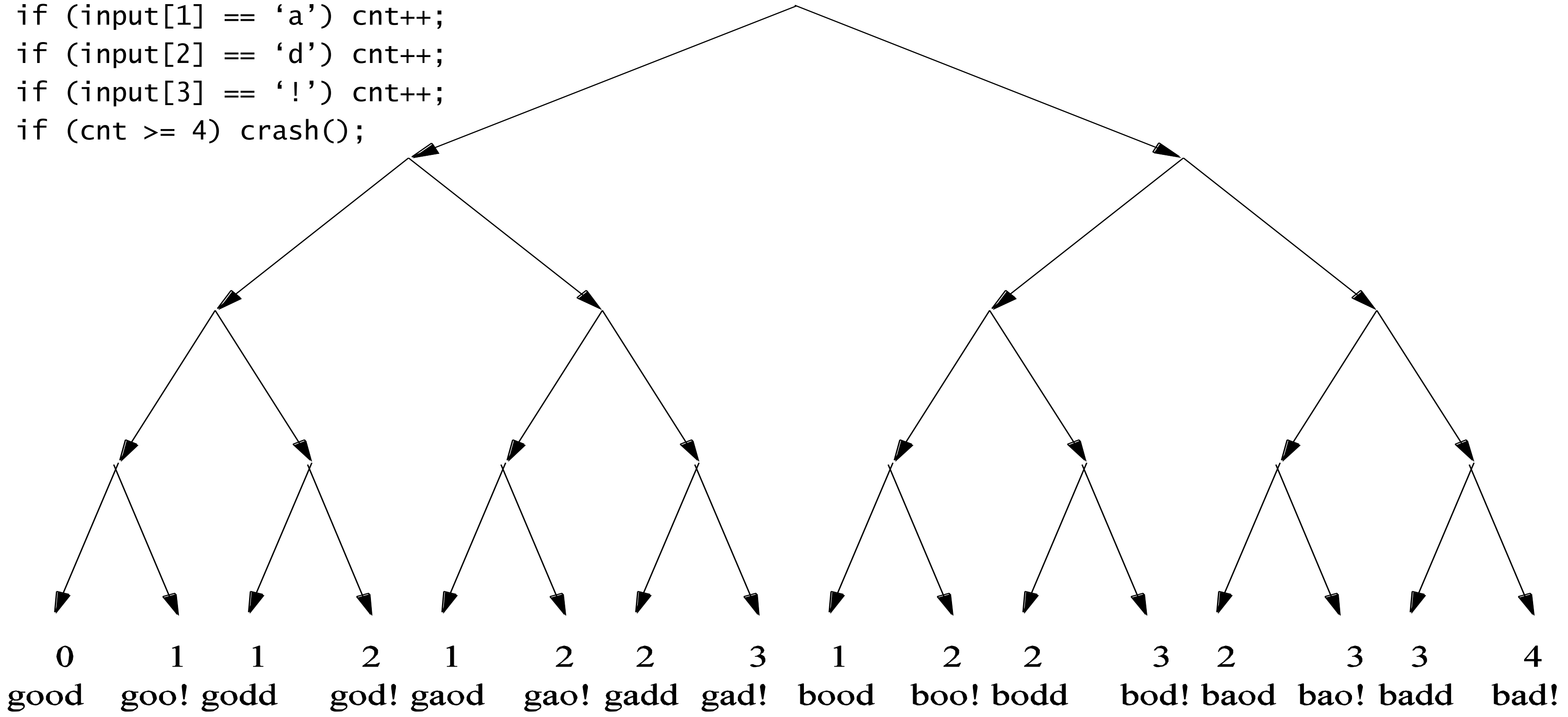
Negate each constraint in path constraint

Solve new constraint → new input

The Search Space

```
void top(char input[4])
{
   int cnt = 0;
   if (input[0] == ‘b’) cnt++;
   if (input[1] == ‘a’) cnt++;
   if (input[2] == ‘d’) cnt++;
   if (input[3] == ‘!’) cnt++;
   if (cnt >= 4) crash();
}
```

If symbolic execution is perfect
and search space is small,
this is verification !



Some Experiments

Most much (100x) bigger than ever tried before!

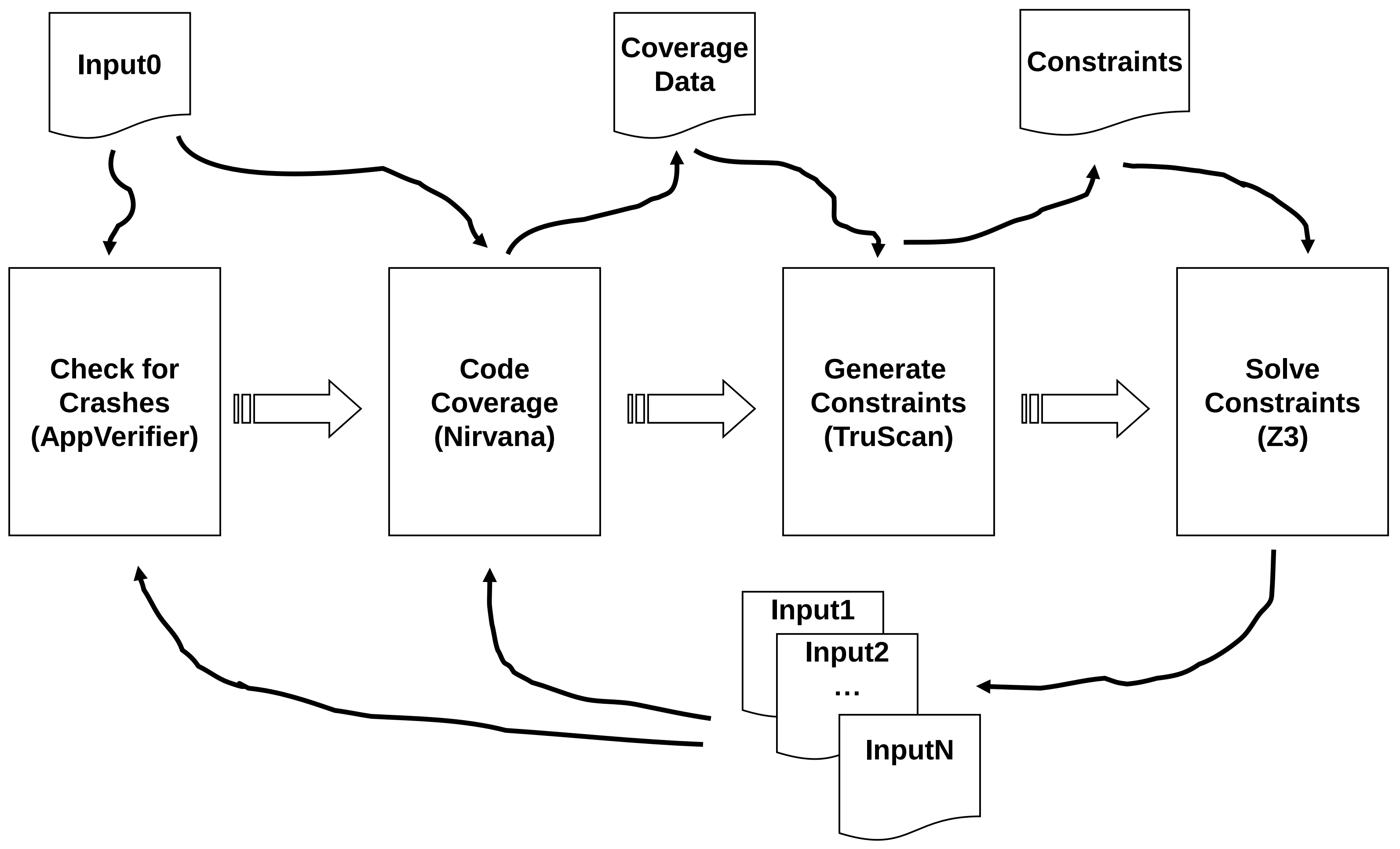
- Seven applications – 10 hours search each

App Tested	#Tests	Mean Depth	Mean #Instr.	Mean Input Size
ANI	11468	178	2,066,087	5,400
Media1	6890	73	3,409,376	65,536
Media2	1045	1100	271,432,489	27,335
Media3	2266	608	54,644,652	30,833
Media4	909	883	133,685,240	22,209
Compressed File Format	1527	65	480,435	634
Excel	3008	6502	923,731,248	45,064

SAGE (Scalable Automated Guided Execution)

- Whitebox fuzzing introduced in SAGE
- Performs symbolic execution of x86 execution traces
  - Builds on Nirvana, iDNA and TruScan for x86 analysis
  - Don't care about language or build process
  - Easy to test new applications, no interference possible
- Can analyse any file-reading Windows applications
- Several optimizations to handle huge execution traces
  - Constraint caching and common subexpression elimination
  - Unrelated constraint optimization
  - Constraint subsumption for constraints from input-bound loops
  - "Flip-count" limit (to prevent endless loop expansions)

SAGE Architecture

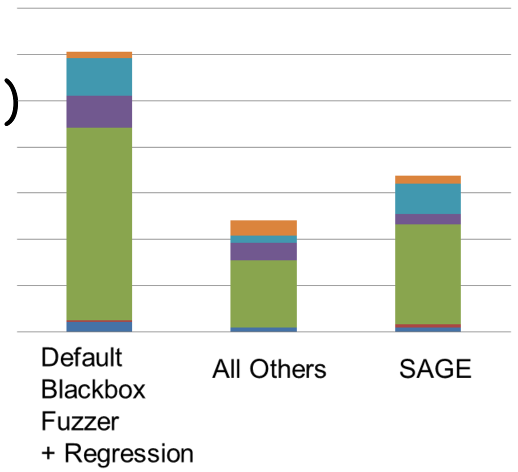


SAGE Results

Since 2007: many new security bugs found
(missed by blackbox fuzzers, static analysis)

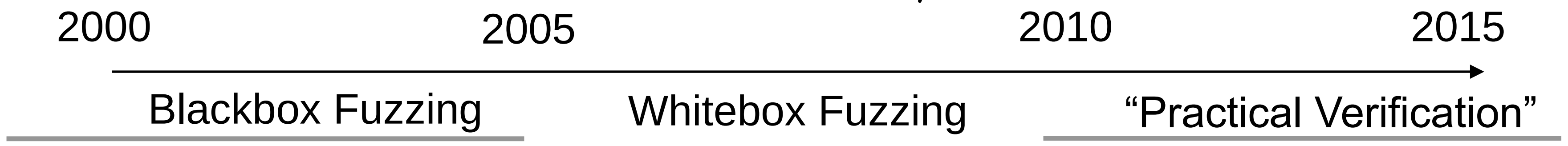
- Apps: image decoders, media players, document processors,...
- Bugs: Write A/Vs, Read A/Vs, Crashes,...
- Many triaged as "security critical, severity 1, priority 1" (would trigger Microsoft security bulletin if known outside MS)
- Example: WEX Security team for Win7
  - Dedicated fuzzing lab with 100s machines
  - 100s apps (deployed on 1 billion+ computers)
  - ~1/3 of all fuzzing bugs found by SAGE !

How fuzzing bugs found (2006-2009) :



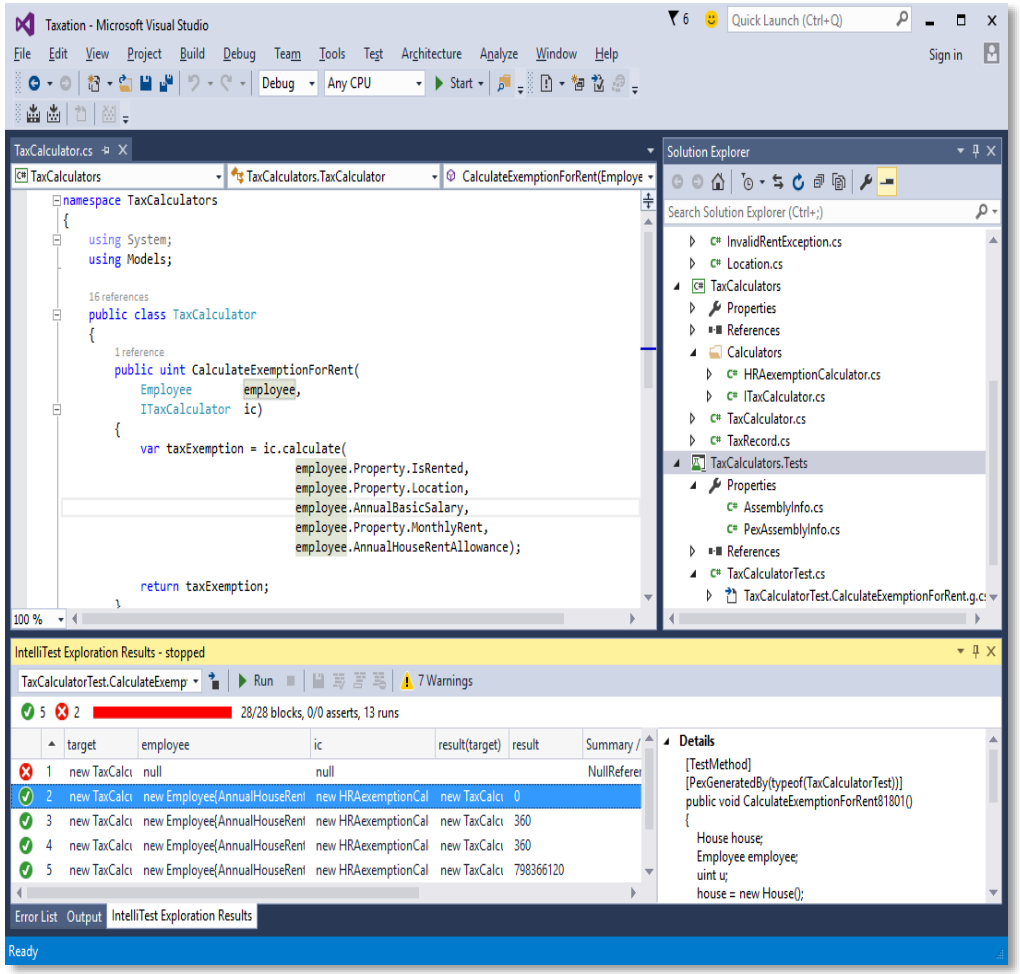
Impact of SAGE (in Numbers)

- 500+ machine-years
  - Runs in the largest dedicated fuzzing lab in the world
  - Largest computational usage ever for any SMT solver
- 100s of apps, 100s of bugs (missed by everything else)
  - Bug fixes shipped quietly (no MSRCs) to 1 Billion+ PCs
  - Millions of dollars saved (for Microsoft and the world)
- "Practical Verification":
  - Eradicate all buffer overflows in all Windows parsers
    - <5 security bulletins in all SAGE-cleaned Win7 parsers, 0 since 2011
    - If nobody can find bugs in P, P is observationally equiv to "verified"!
    - Reduce costs & risks for Microsoft, increase those for Black Hats



Pex & Moles (MSR): Unit Testing for .NET

- User specifies a "parameterized unit test", leverages code contracts
- Pex automatically generates tests (using modified DART algorithm)
- Moles: framework for mock-object creation
- See rise4fun.com and pex4fun.com
- Over 40,000 downloads
- Re-named IntelliTest and Fakes in VisualStudio'2015 ->



```
namespace TaxCalculators
{
    using System;
    using Models;

    16 references
    public class TaxCalculator
    {
        1 reference
        public uint CalculateExemptionForRent(
            Employee        employee,
            ITaxCalculator  ic)
        {
            var taxExemption = ic.calculate(
                                    employee.Property.IsRented,
                                    employee.Property.Location,
                                    employee.AnnualBasicSalary,
                                    employee.Property.MonthlyRent,
                                    employee.AnnualHouseRentAllowance);

            return taxExemption;
        }
```

IntelliTest Exploration Results - stopped

✓ 5 ✗ 2 28/28 blocks, 0/0 asserts, 13 runs

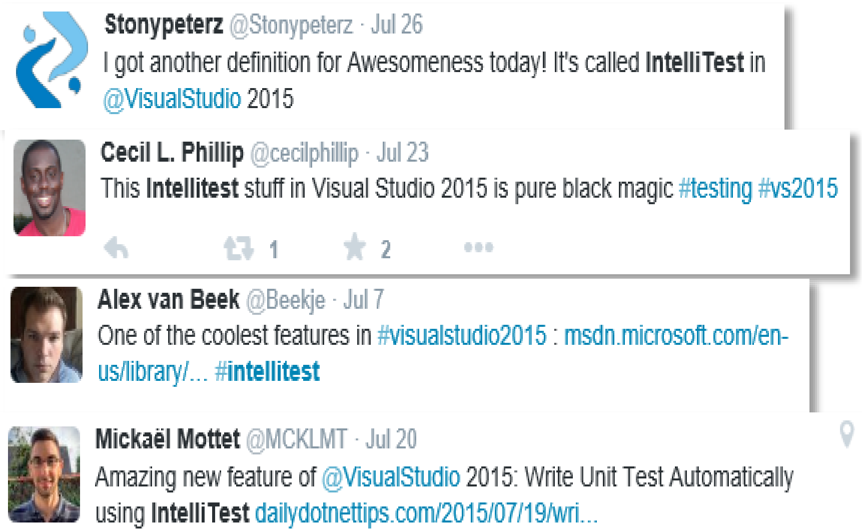
		target	employee	ic	result(target)	result	Summary /
✗	1	new TaxCalcu	null	null			NullReferen
✓	2	new TaxCalcu	new Employee{AnnualHouseRent	new HRAexemptionCal	new TaxCalcu	0	
✓	3	new TaxCalcu	new Employee{AnnualHouseRent	new HRAexemptionCal	new TaxCalcu	360	
✓	4	new TaxCalcu	new Employee{AnnualHouseRent	new HRAexemptionCal	new TaxCalcu	360	
✓	5	new TaxCalcu	new Employee{AnnualHouseRent	new HRAexemptionCal	new TaxCalcu	798366120	

Details

```
[TestMethod]
[PexGeneratedBy(typeof(TaxCalculatorTest))]
public void CalculateExemptionForRent81801()
{
  House house;
  Employee employee;
  uint u;
  house = new House();
```

Feedback (2015)…

- Very early days for Visual Studio 2015 upgrade cycle
  - Just 2 months since shipping (7/20/2015)
- Favourable response from early adopters
  - IntelliTest NUnit Extension - 1557 downloads
  - IntelliTest xUnit.net Extension - 1239 downloads
- Positive sentiment on twitter.
- Growing "asks" on uservoice already.

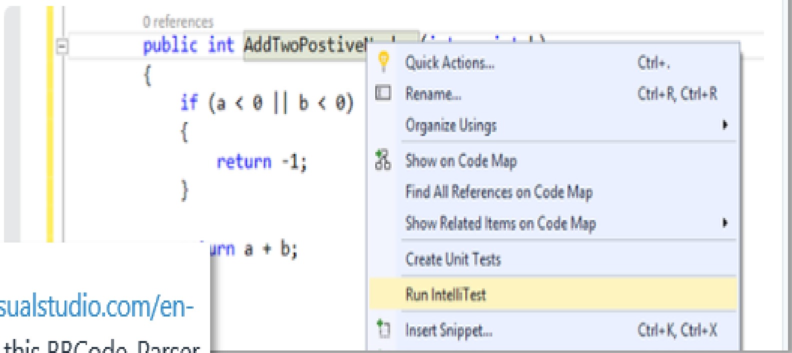


Stonypeterz @Stonypeterz · Jul 26
I got another definition for Awesomeness today! It's called **IntelliTest** in @VisualStudio 2015

Cecil L. Phillip @cecilphillip · Jul 23
This **Intellitest** stuff in Visual Studio 2015 is pure black magic #testing #vs2015
1 2

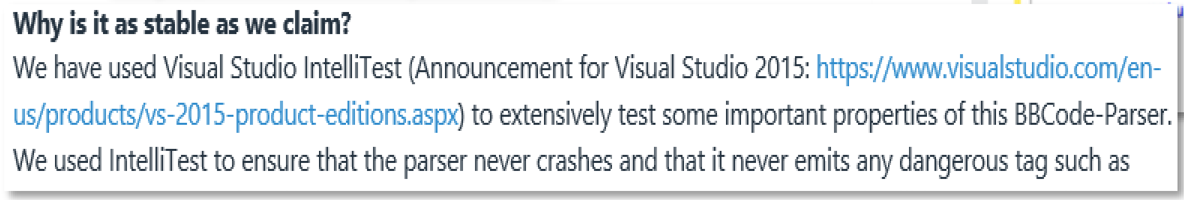
Alex van Beek @Beekje · Jul 7
One of the coolest features in #visualstudio2015 : msdn.microsoft.com/en-us/library/... **#intellitest**

Mickaël Mottet @MCKLMT · Jul 20
Amazing new feature of @VisualStudio 2015: Write Unit Test Automatically using **IntelliTest** dailydotnettips.com/2015/07/19/wri...



```
0 references
public int AddTwoPostive
{
    if (a < 0 || b < 0)
    {
        return -1;
    }
    urn a + b;
```

From http://bbcode.codeplex.com/



Why is it as stable as we claim?

We have used Visual Studio IntelliTest (Announcement for Visual Studio 2015: https://www.visualstudio.com/en-us/products/vs-2015-product-editions.aspx) to extensively test some important properties of this BBCode-Parser. We used IntelliTest to ensure that the parser never crashes and that it never emits any dangerous tag such as

Code Hunt

www.codehunt.com

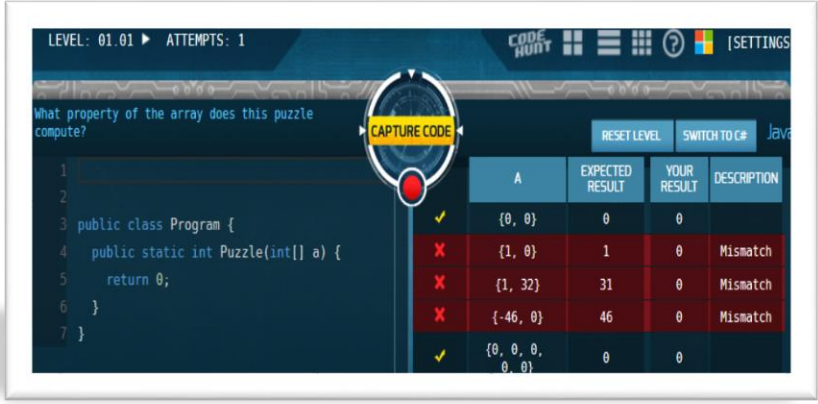
Age: 13+

Platform: via a web browser

Languages: C# and Java

Description: A game for practicing programming and running coding contests. Successive puzzles are presented with test cases only, no specifications. Players have to first work out the pattern and then code the answer.

Teaches: Imperative programming
Testing skills



What property of the array does this puzzle compute?

```


public class Program {
  public static int Puzzle(int[] a) {
    return 0;
  }
}
```

	A	EXPECTED RESULT	YOUR RESULT	DESCRIPTION
✓	{0, 0}	0	0	
✗	{1, 0}	1	0	Mismatch
✗	{1, 32}	31	0	Mismatch
✗	{-46, 0}	46	0	Mismatch
✓	{0, 0, 0, 0, 0}	0	0	

Adoption: Over 300,000 users (8,000 a week, 90% returning users) world-wide

Tech: Uses Pex to check solutions, open source data available for analysis

Conclusion: Dynamic Software Model Checking

$$\begin{tikzcd}
\text{Modeling languages} \arrow[r, "\text{state-space exploration}"] & \text{Model checking} \arrow[d, "\text{adaptation}"] \\
\text{Programming languages} \arrow[u, "\text{abstraction}"] \arrow[r, "\text{state-space exploration}"] & \text{Systematic testing}
\end{tikzcd}$$

(SLAM, Bandera, FeaVer, BLAST,…)

Dynamic

Concurrency: **VeriSoft, JPF, CMC, CHESS, Cuzz,…**
Data inputs: **DART, EXE, SAGE, Pex,…**

- Dynamic Software Model Checking
  - Concurrency: equivalence classes of program executions using partial-order reduction
  - Data inputs: equivalence classes of program executions using (dynamic) symbolic execution

Conclusion

- Dynamic Software Model Checking
  - Scales to industrial-size software
  - Dozens of tools and applications over the last 20 years
    - In industry and academia
  - Can find bugs that traditional testing cannot find !
  - Significant impact
    - Ex: SAGE found bugs in Windows and Office apps used by billions
    - And many more examples of applications !
  - Yet not enough impact !
    - Need better algorithms and tools (especially for concurrency!)
    - Need better scalability, precision, automation,… usefulness !
    - Need more "killer apps" and users