
Software Model Checking: Searching for Computations in the Abstract or the Concrete

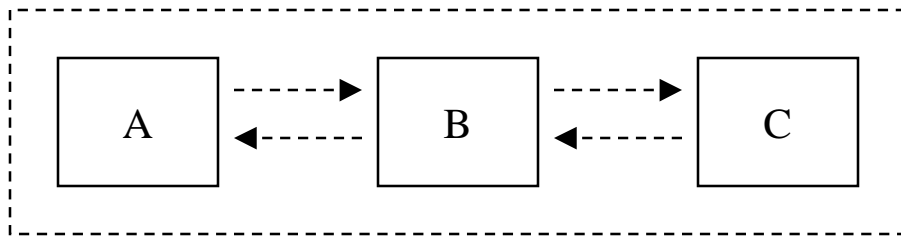
Patrice Godefroid

Bell Laboratories, Lucent Technologies

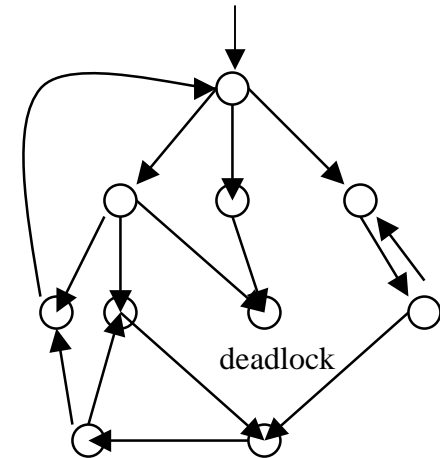
Overview

- Goal: an overview of software model checking
 - Past and current efforts
 - Future trends
- A discussion of the forces in play
 - Validation versus Falsification
 - Static (abstract) versus Dynamic (concrete) Analysis, and their [integration](#)
 - See paper in IFM'2005 Proc. for more (co-authored with Nils Klarlund)
- Disclaimer:
 - a personal view of where the field started and where it is currently going
 - emphasis on technical ideas, not references
 - emphasis on what influenced the speaker, not a fully exhaustive survey

“Model Checking”



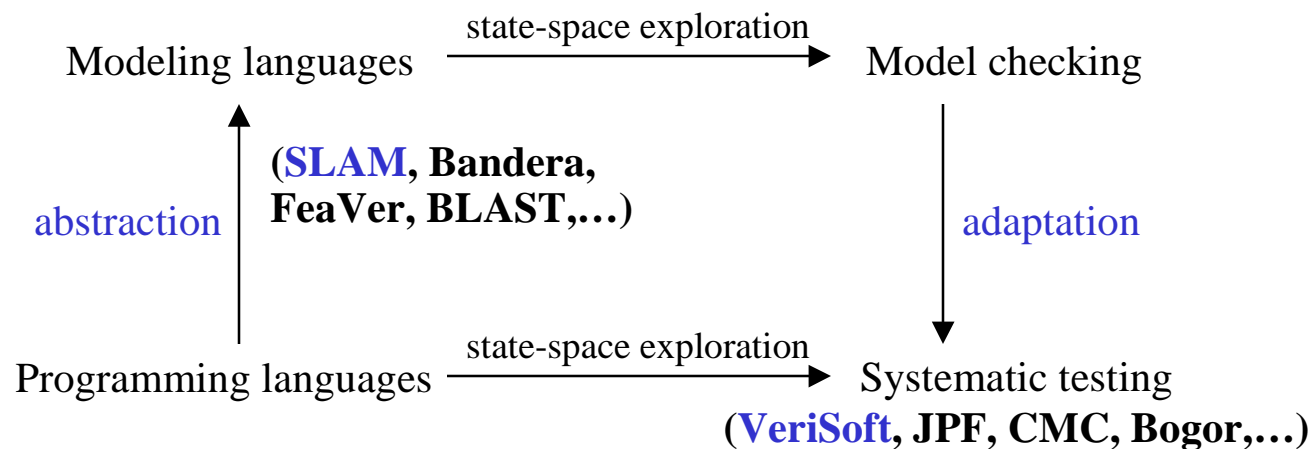
Each component is modeled by a FSM.



- 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 Hardware, SPIN for Software, etc.)
- Once thoroughly checked, models can be compiled and used as the core of the implementation (examples: SDL, VFSM, etc.)

Model Checking of Software

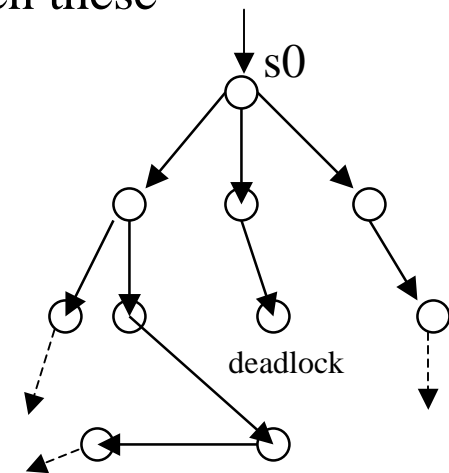
- Challenge: how to apply model checking to analyze **software**?
 - “Real” programming languages (e.g., C, C++, Java),
 - “Real” size (e.g., 100,000’s lines of code).
- Two main approaches to software model checking:



Dynamic Approach: Systematic Testing (VeriSoft)

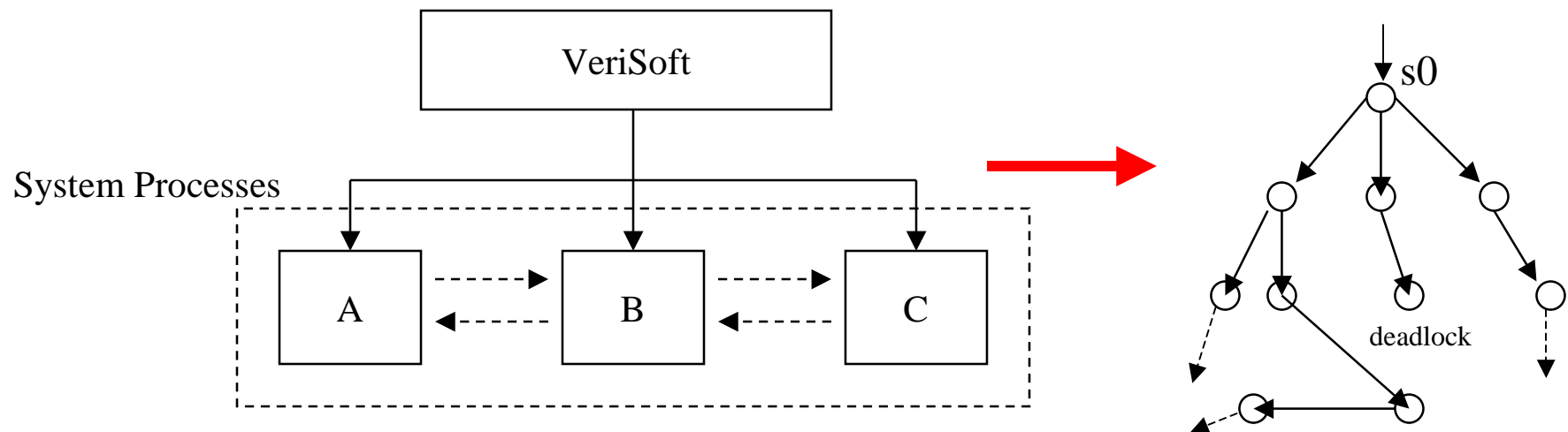
- State Space (Dynamic Semantics)= “product of (Unix) processes”
 - Processes communicate by executing operations on com. objects
 - Operations on com. objects are visible, other operations are invisible
 - Only executions of visible operations may be blocking
 - The system is in a global state when 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: (safety properties only!)
 - 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 to a queue, and Process 2 receives from it. An assertion violation is indicated by a red vertical line.
- VeriSoft Simulator - (Pruned) State Space View:** A state space graph showing nodes representing different states. The initial state is highlighted, and a path of states is shown leading to an assertion violation. The key indicates: Assertion Violations: 1, Deadlocks: 0, Aborts: 0.
- VeriSoft Simulator:** A dialog box with the text "Assertion violation!" and a "Dismiss" button.
- VeriSoft Simulator - Process 2:** A window showing the execution flow for Process 2, with buttons for Step, Next, Continue, Print, and Quit.
- VeriSoft Simulator - Process 1:** A window showing the execution flow for Process 1, with buttons for Step, Next, Continue, Print, and Quit.
- VeriSoft Simulator - State Space Filter:** A window with a "Text Regular Expression:" field and "Match", "Clear", and "Quit" buttons. It lists labels and processes to be filtered.
- Terminal Window:** Shows the command prompt with the following text:

```
pts/2 /home/god/verisoft/examples/ac-controller  
comeback $ verisoft main.c -simul error1.path  
gcc -I/home/god/verisoft/bin /home/god/verisoft/bin/verisoft_simul_Sun05_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.
```


VeriSoft - Summary

- VeriSoft is the first software model checker for general-purpose programming languages such as C and C++ [POPL97,Godefroid]
- 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 to make this approach tractable in the 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
- Subsequent related tools: JPF (NASA; Java, stateful via instrumented JVM), CMC (Stanford; C, stateful, symmetry reduction), Bogor (Kansas U.), etc.

VeriSoft Users and Applications

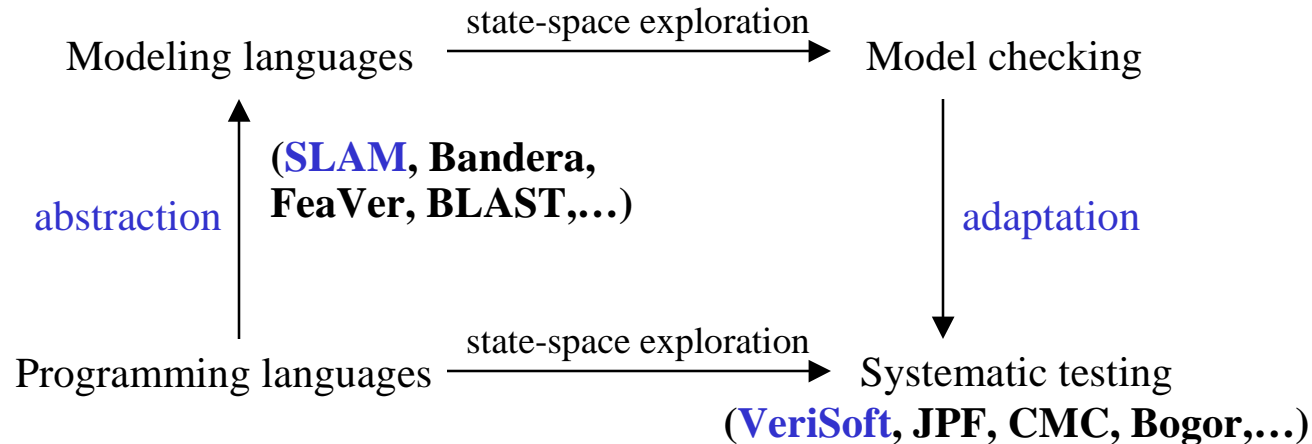
- Development of research prototype started in 1996
- VeriSoft 2.0 available outside Lucent since January 1999:
 - 100's of licenses in 25+ countries, in industry and academia
 - Free download at <http://www.bell-labs.com/projects/verisoft>
- Examples of applications in Lucent:
 - 4ESS Heart-Beat-Monitor unit testing and debugging (telephone switch maintenance) [ISSTA'98]
 - 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) [ICSE'2002]

Discussion (Strengths and Limitations)

- VeriSoft (like model checking) is not a panacea
 - Limited by state-explosion...
 - 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?”

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:

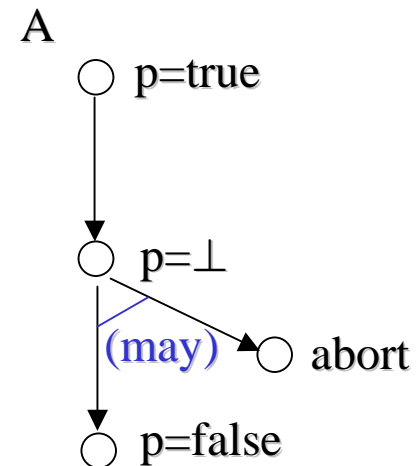


Static Approach: Automatic Abstraction (SLAM)

- “Abstract-Check-Refine” Loop:
 1. Abstract: generate a (may) abstraction via static program analysis
 - Ex: predicate abstraction and boolean program
 2. Check: “model check” the abstraction
 3. Refine: map abstract error traces back to code, or refine the abstraction (e.g., by adding predicates); goto 1

```
Program P( ) {  
  int x = 1;  
  x = h(x);  
  if (odd(x))  
    abort(); // error!  
  x = 0;  
}
```

Predicate abstraction
p: “x is odd”



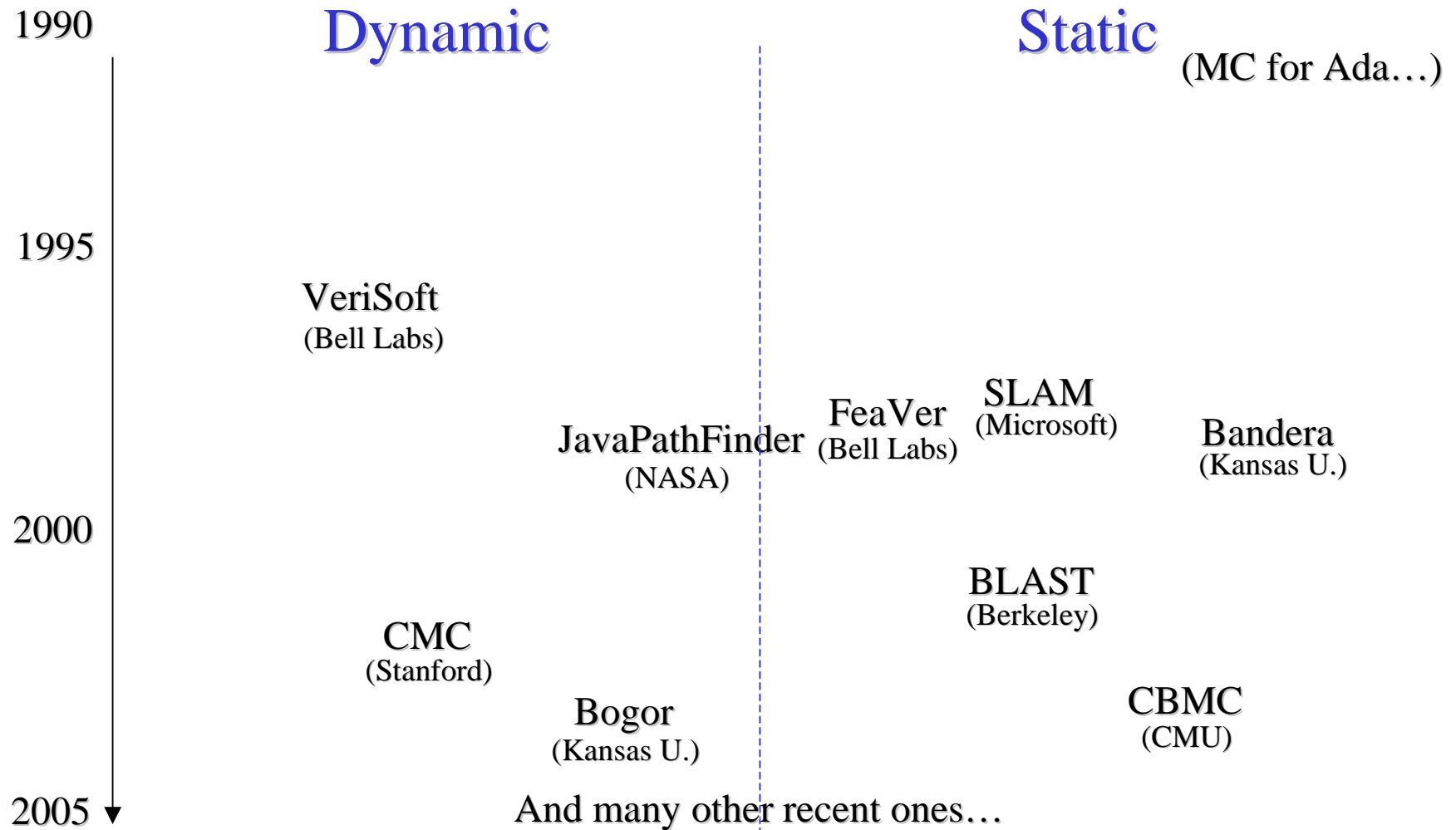
Main Ideas and Issues

1. Abstract: extract a “model” out of concrete program via static analysis
 - Which programming languages are supported? ((subset of) C, Java, Ada, Domain-Specific Language?)
 - Additional assumptions? (Pointers? Recursion? Concurrency?...)
 - What is the target modeling language? ((C)(E)FSMs, PDAs,...)
 - Can/must the abstraction process be guided by the user? How?
2. Model check the abstraction
 - What properties can be checked? (Safety? Liveness?,...)
 - How to model the environment? (Closed or open system ?...)
 - Which model-checking algorithm? (New algos for PDAs, use SAT solvers...)
 - Is the abstraction “conservative”? (I.e., is the static analysis “sound”?)
3. Map abstract counter-examples back to code, or refine the abstraction
 - Behaviors violating the property may have been introduced during Step 1
 - How to map scenarios leading to errors back to the code?
 - When an error trace is spurious, how to refine the abstraction?

Lots of Recent Work...

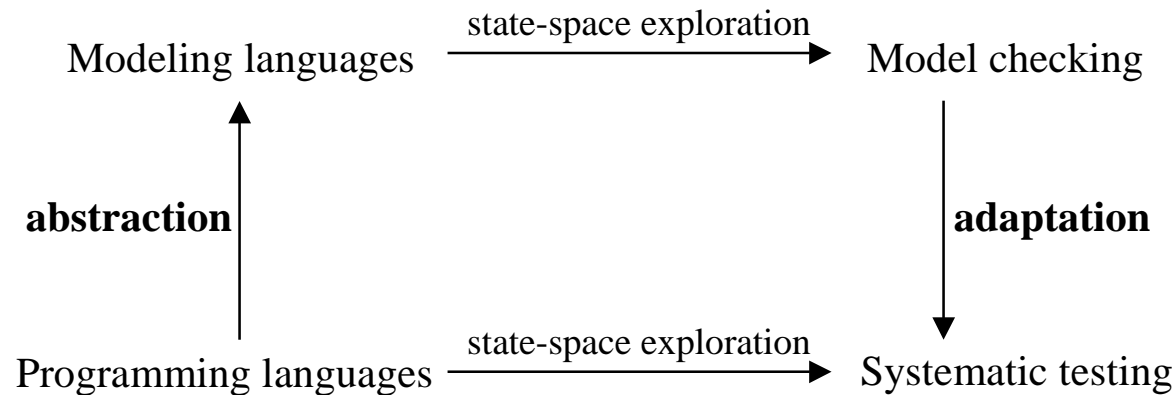
- Examples of tools:
 - SLAM (Microsoft): see previous slides; now part of Microsoft Windows device-driver development toolkit
 - Bandera (Kansas U.): Java to SPIN/SMV/* using user-guided abstraction mapping and slicing/abstract-interpretation/*
 - FeaVer (Bell Labs): C to SPIN using user-specified abstraction mapping
 - BLAST (Berkeley): similar to SLAM but “lazy abstraction refinement”
 - Etc! (+ Tools for static analysis of concurrent programs, Ada, etc.)
- Examples of frameworks: (automatic abstraction refinement)
 - [Graf,Saidi,...], [Clarke,Grumberg,Jha,...], [Ball,Rajamani,Podelski,...], [Dill,Das,...], [Khurshan,Namjoshi,...], [Dwyer,Pasareanu,Visser,...], [Bruns,Godefroid,Huth,Jagadeesan,Schmidt...], [Henzinger, Jhala, Majumdar,...], and many more!

Software Model Checking Tools (for C,C++,Java...)



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
- May produce spurious counterexamples (unsound bugs)
- Can prove correctness (complete) in theory (but not in practice...)

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 bugs)
- Provide a complete state-space coverage up to some depth only (typically incomplete)

Model Checking of Software: What Next?

- A new generation of software model checkers combining static and dynamic analysis is coming up...
- Motivation: take the best of both approaches (precision of dynamic analysis AND efficiency of static analysis)
- Example: DART (Directed Automated Random Testing)
 - See [PLDI'2005], joint work done at Bell Labs with Nils Klarlund and Koushik Sen (summer intern from UIUC)
 - Can be viewed as extending the VeriSoft approach to data nondeterminism (see also [PLDI'98, Colby-Godefroid-Jagadeesan] for an earlier attempt)
 - Uses static program analysis and symbolic execution techniques (including theorem proving) for systematic test-input generation and execution
 - Just one way to combine static and dynamic analysis for software model checking...

DART: Directed Automated Random Testing

1. **Automated** extraction of program interface from source code
 2. Generation of test driver for **random** testing through the interface
 3. Dynamic test generation to **direct** executions along alternative program paths
- Together: (1)+(2)+(3) = DART
 - DART can detect program crashes and assertion violations
 - Any program that compiles can be run and tested this way:

No need to write any test driver or harness code!
 - (Pre- and post-conditions can be added to generated test-driver)

Example (C code)

```
int double(int x) {  
    return 2 * x;  
}
```



(1) Interface extraction:

- parameters of top-level function
- external variables
- return values of external functions



(2) Generation of test driver for random testing:

```
void test_me(int x, int y) {  
    int z = double(x);  
    if (z==y) {  
        if (y == x+10)  
            abort(); /* error */  
    }  
}
```

```
main(){  
    int tmp1 = randomInt();  
    int tmp2 = randomInt();  
    test_me(tmp1,tmp2);  
}
```



Closed (self-executable) program that can be run

Problem: probability of reaching `abort()` is extremely low!

DART Step (3): Directed Search

```
main(){
```

```
    int t1 = randomInt();
```

```
    int t2 = randomInt();
```

```
    test_me(t1,t2);
```

```
}
```

```
int double(int x) {return 2 * x; }
```

```
void test_me(int x, int y) {
```

```
    int z = double(x);
```

```
    if (z==y) {
```

```
        if (y == x+10)
```

```
            abort(); /* error */
```

```
    }
```

```
}
```

Concrete
Execution

Symbolic
Execution

Path
Constraint

x = 36, y = 99

create symbolic
variables x, y

DART Step (3): Directed Search

```
main(){
```

```
    int t1 = randomInt();
```

```
    int t2 = randomInt();
```

```
    test_me(t1,t2);
```

```
}
```

```
int double(int x) {return 2 * x; }
```

```
void test_me(int x, int y) {
```

```
    int z = double(x);
```

```
    ← if (z==y) {
```

```
        if (y == x+10)
```

```
            abort(); /* error */
```

```
        }
```

```
    }
```

Concrete
Execution

Symbolic
Execution

Path
Constraint

x = 36, y = 99,
z = 72

create symbolic
variables x, y
z = 2 * x

DART Step (3): Directed Search

```
main(){
  int t1 = randomInt();
  int t2 = randomInt();
  test_me(t1,t2);
}
int double(int x) {return 2 * x; }

void test_me(int x, int y) {
  int z = double(x);
  if (z==y) {
    if (y == x+10)
      abort(); /* error */
  }
}
```

Concrete Execution

Symbolic Execution

Path Constraint

Solve: $2 * x == y$

Solution: $x = 1, y = 2$

create symbolic variables x, y

$2 * x != y$

$x = 36, y = 99,$
 $z = 72$

$z = 2 * x$

DART Step (3): Directed Search

```
main(){
  int t1 = randomInt();
  int t2 = randomInt();
  test_me(t1,t2);
}
int double(int x) {return 2 * x; }
void test_me(int x, int y) {
  int z = double(x);
  if (z==y) {
    if (y == x+10)
      abort(); /* error */
  }
}
```

Concrete Execution

x = 1, y = 2

Symbolic Execution

create symbolic variables x, y

Path Constraint

DART Step (3): Directed Search

```
main(){
    int t1 = randomInt();
    int t2 = randomInt();
    test_me(t1,t2);
}
int double(int x) {return 2 * x;}

void test_me(int x, int y) {
    int z = double(x);
    ← if (z==y) {
        if (y == x+10)
            abort(); /* error */
    }
}
```

Concrete
Execution

Symbolic
Execution

Path
Constraint

x = 1, y = 2, z = 2

create symbolic
variables x, y

z = 2 * x

DART Step (3): Directed Search

```

main(){
    int t1 = randomInt();
    int t2 = randomInt();
    test_me(t1,t2);
}

int double(int x) {return 2 * x; }

void test_me(int x, int y) {
    int z = double(x);
    if (z==y) {
        if (y == x+10)
            abort(); /* error */
    }
}
    
```

Concrete Execution

Symbolic Execution

Path Constraint

$x = 1, y = 2, z = 2$

create symbolic variables x, y

$z = 2 * x$

$2 * x == y$

DART Step (3): Directed Search

```

main(){
    int t1 = randomInt();
    int t2 = randomInt();
    test_me(t1,t2);
}

int double(int x) {return 2 * x;}

void test_me(int x, int y) {
    int z = double(x);
    if (z==y) {
        if (y == x+10)
            abort(); /* error */
    }
}
    
```

Concrete Execution

Symbolic Execution

Path Constraint

Solve: $(2 * x == y) \wedge (y == x + 10)$

Solution: $x = 10, y = 20$

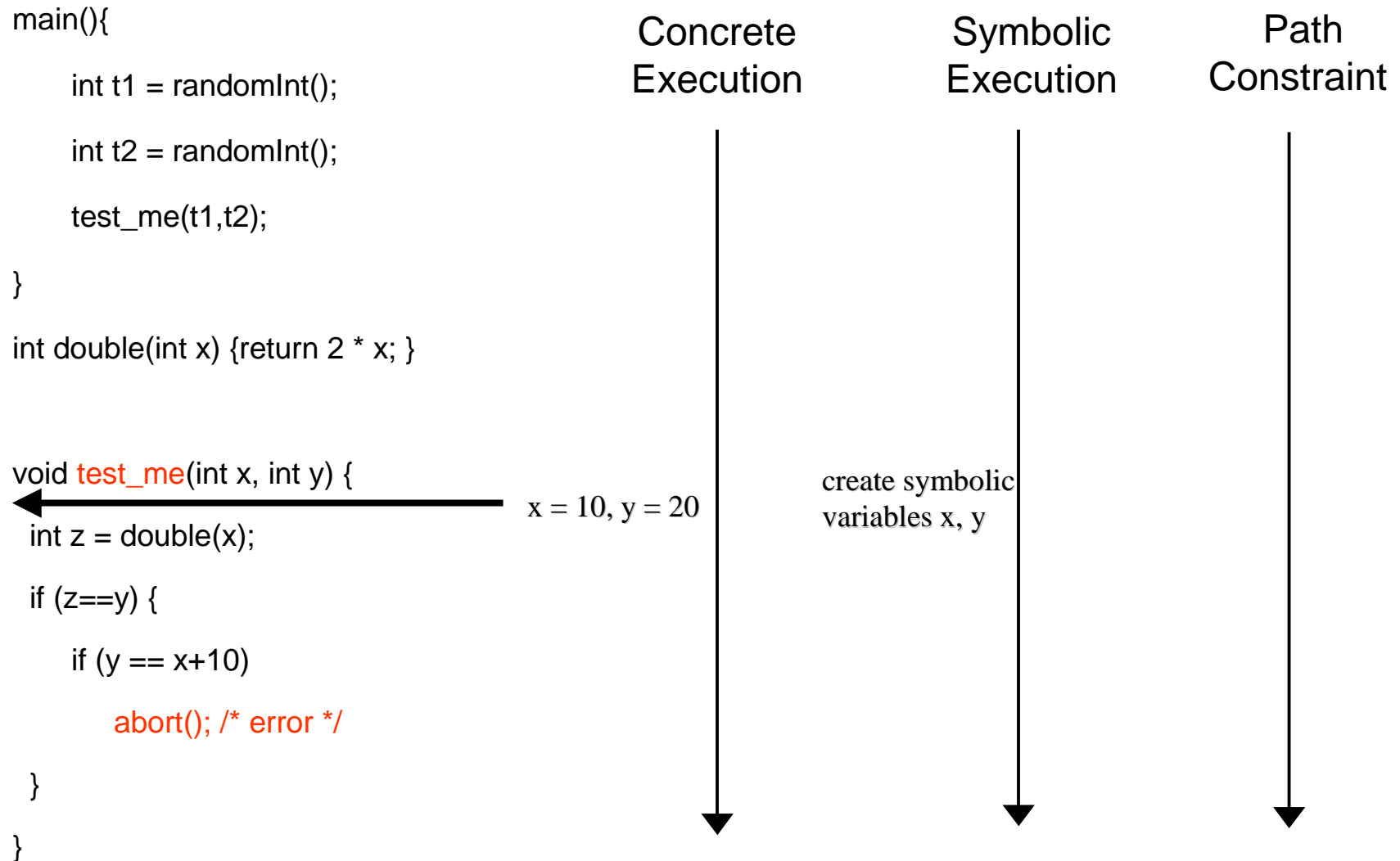
create symbolic variables x, y

$2 * x == y$
 $y != x + 10$

$x = 1, y = 2, z = 2$

$z = 2 * x$

DART Step (3): Directed Search



DART Step (3): Directed Search

```
main(){
```

```
    int t1 = randomInt();
```

```
    int t2 = randomInt();
```

```
    test_me(t1,t2);
```

```
}
```

```
int double(int x) {return 2 * x; }
```

```
void test_me(int x, int y) {
```

```
    int z = double(x);
```

```
    ← if (z==y) {
```

```
        if (y == x+10)
```

```
            abort(); /* error */
```

```
        }
```

```
    }
```

Concrete
Execution

Symbolic
Execution

Path
Constraint

x = 10, y = 20, z = 20

create symbolic
variables x, y

z = 2 * x

DART Step (3): Directed Search

```
main(){
  int t1 = randomInt();
  int t2 = randomInt();
  test_me(t1,t2);
}
int double(int x) {return 2 * x;}

void test_me(int x, int y) {
  int z = double(x);
  if (z==y) {
    if (y == x+10)
      abort(); /* error */
  }
}
```

Concrete
Execution

Symbolic
Execution

Path
Constraint

x = 10, y = 20, z = 20

create symbolic
variables x, y

z = 2 * x

2 * x == y

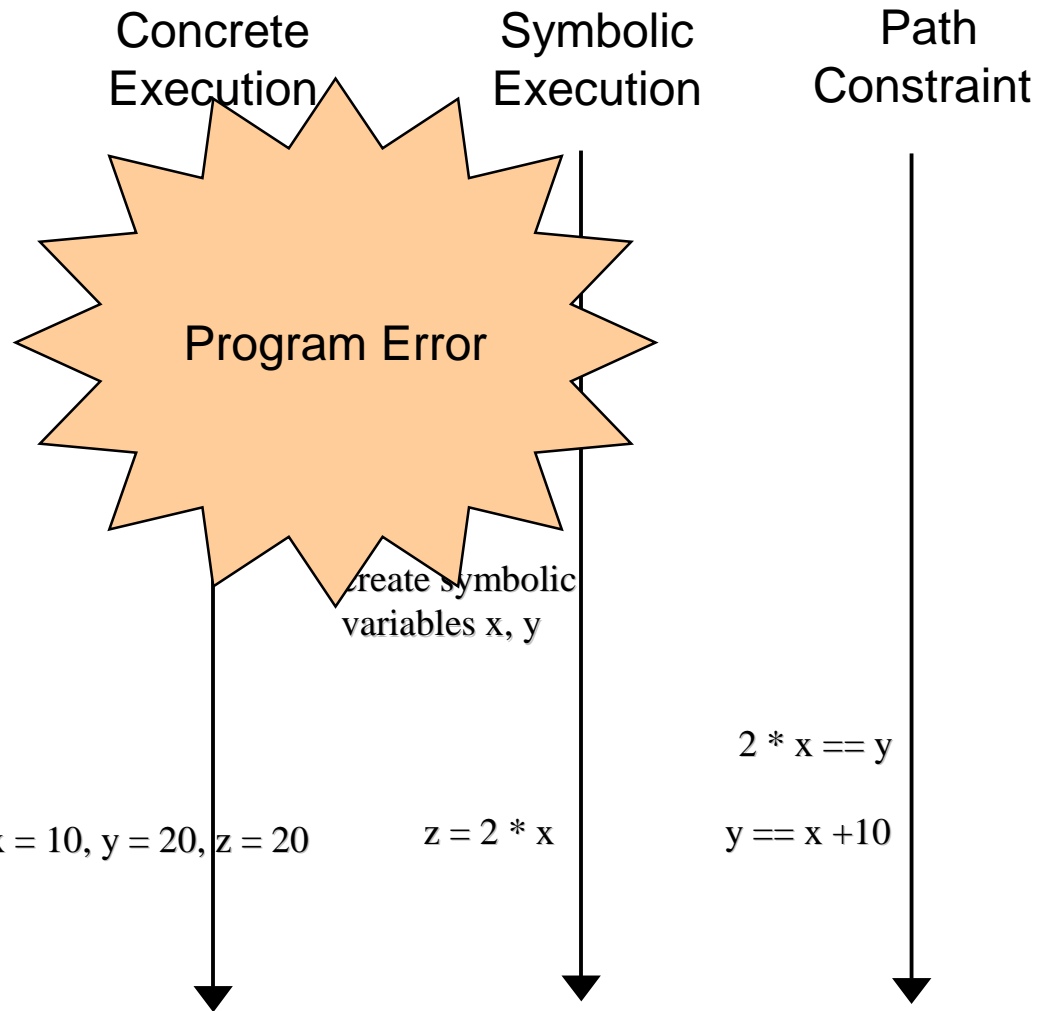
DART Step (3): Directed Search

```

main(){
    int t1 = randomInt();
    int t2 = randomInt();
    test_me(t1,t2);
}

int double(int x) {return 2 * x;}

void test_me(int x, int y) {
    int z = double(x);
    if (z==y) {
        if (y == x+10)
            abort(); /* error */
    }
}
    
```



Directed Search: Summary

- Dynamic test generation to **direct** executions along alternative program paths
 - collect symbolic constraints at branch points (whenever possible)
 - negate one constraint at a branch point to take other branch (say **b**)
 - call constraint solver with new path constraint to generate new test inputs
 - next execution driven by these new test inputs to take alternative branch **b**
 - check with dynamic instrumentation that branch **b** is indeed taken
- Repeat this process until all execution paths are covered
 - May never terminate!
- Significantly improves code coverage vs. pure random testing

Novelty: Simultaneous Concrete & Symbolic Executions

```
void foo(int x,int y){  
    int z = x*x*x; /* could be z = h(x) */  
    if (z == y) {  
        abort(); /* error */  
    }  
}
```

- Assume we can reason about linear constraints only
- Initially $x = 3$ and $y = 7$ (randomly generated)
- Concrete $z = 27$, but symbolic $z = x*x*x$
 - Cannot handle symbolic value of z !
 - Stuck?

Novelty: Simultaneous Concrete & Symbolic Executions

```
void foo(int x,int y){  
    int z = x*x*x; /* could be z = h(x) */  
    if (z == y) {  
        abort(); /* error */  
    }  
}
```

Replace symbolic expression by **concrete value** when symbolic expression becomes **unmanageable** (e.g. non-linear)

NOTE: whenever symbolic execution is stuck, **static analysis** becomes imprecise!

- Assume we can reason about linear constraints only
- Initially $x = 3$ and $y = 7$ (randomly generated)
- Concrete $z = 27$, but symbolic $z = x*x*x$
 - Cannot handle symbolic value of z !
- Stuck?
 - **NO!** Use concrete value $z = 27$ and proceed...
- Take else branch with constraint $27 \neq y$
- Solve $27 = y$ to take then branch
- Execute next run with $x = 3$ and $y = 27$
- DART finds the error!

Comparison with Static Analysis

```
1 foobar(int x, int y){
2   if (x*x*x > 0){
3     if (x>0 && y==10){
4       abort(); /* error */
5     }
6   } else {
7     if (x>0 && y==20){
8       abort(); /* error */
9     }
10  }
11 }
```

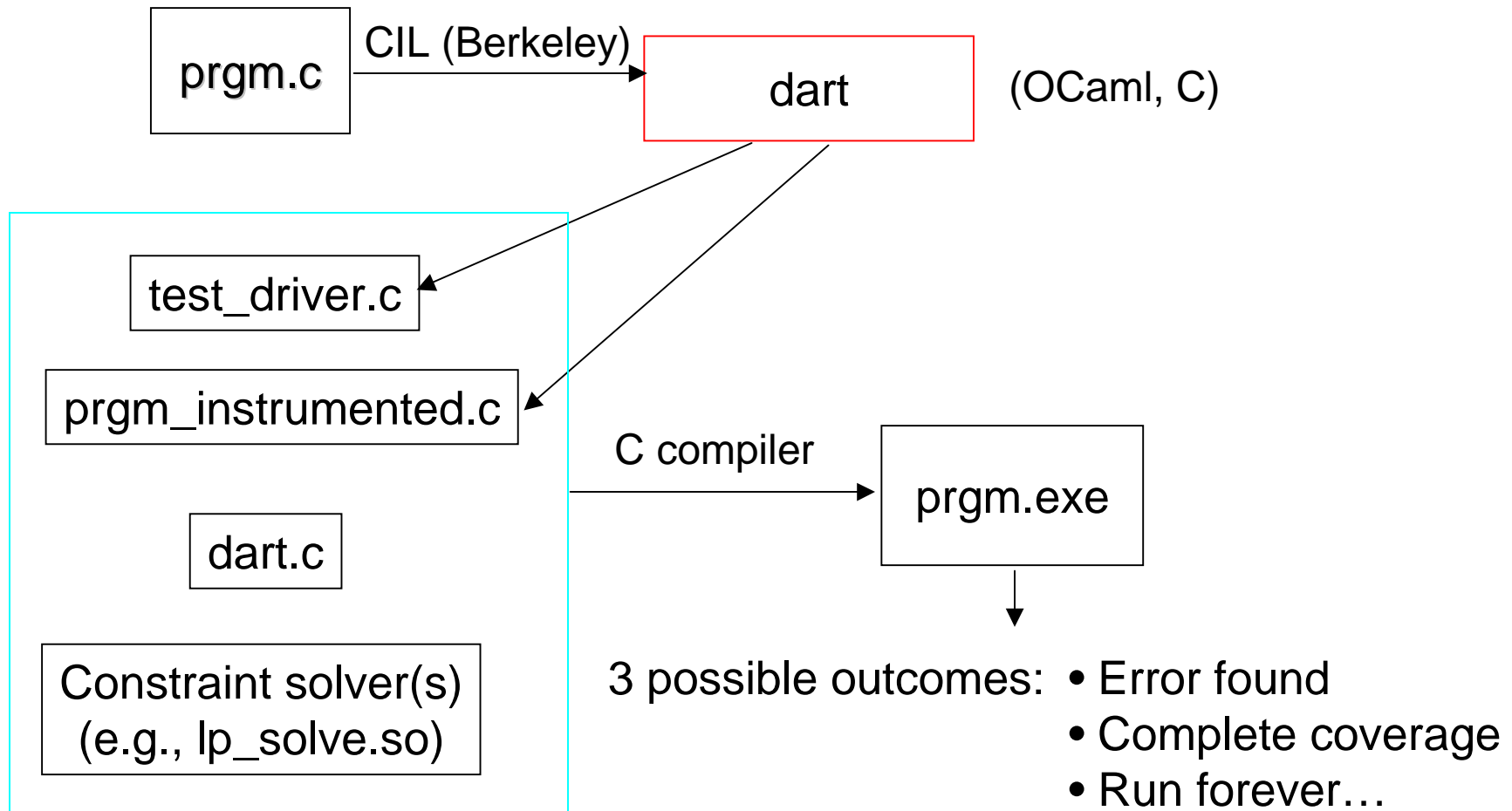
- Symbolic execution is stuck at line 2...
- Static analysis tools will conclude that **both** aborts **may** be reachable
 - “Sound” tools will report both, and thus one false alarm
 - “Unsound” tools will report “no bug found”, and miss a bug
- Static-analysis-based test generation techniques are also helpless here...
- In contrast, DART finds the only error (line 4) with high probability
- Unlike static analysis, **all bugs** reported by DART are guaranteed to be **sound**

Other Advantages of Dynamic Analysis

```
1 struct foo { int i; char c; }
2
3 bar (struct foo *a) {
4     if (a->c == 0) {
5         *((char *)a + sizeof(int)) = 1;
6         if (a->c != 0) {
7             abort();
8         }
9     }
10 }
```

- Dealing with dynamic data is easier with concrete executions
- Due to limitations of alias analysis, static analysis tools cannot determine whether “a->c” has been rewritten
 - “the abort **may** be reachable”
- In contrast, DART finds the error easily (by solving the linear constraint a->c == 0)
- In summary, all bugs reported by DART are guaranteed to be sound!
- But DART may not terminate...

DART for C: Implementation Details



Experiments: NS Authentication Protocol

- Tested a C implementation of a security protocol (Needham-Schroeder) with a known attack
 - About 400 lines of C code; experiments on a Linux 800Mz P-III machine
 - DART takes 57 seconds (9,926 runs) to discover a full attack, with a realistic (Dolev-Yao) intruder model
 - In contrast, VeriSoft could not find this attack in 24 hours (albeit with a different, concurrent and nondeterministic, Dolev-Yao intruder model)
 - Also, the static software model checker BLAST reports a spurious error after 6 minutes of search (due to imprecision of current alias analysis used), and does not find the attack
- DART found a new bug in this C implementation of Lowe's fix to the NS protocol (bug confirmed by the code's author)

A Larger Application: oSIP

- Open Source SIP library (Session Initiation Protocol)
 - 30,000 lines of C code (version 2.0.9), 600 externally visible functions

- Results:

Attack: send a packet of size 2.5 MB (cygwin) with no 0 or “|” character

- DART crashed 65% of the externally visible functions within 1000 runs
- Most of these due to missing(?) NULL-checks for pointers...
- Analysis of results for oSIP parser revealed a simple attack to crash it!

oSIP version 2.0.9 (August 2004)

```
int osip_message_parse (osip_message_t * sip,  
                        const char *buf)
```

```
{ [ ... ]
```

```
char *tmp;
```

```
tmp = alloca (strlen (buf) + 2);
```

```
osip_strncpy (tmp, buf, strlen (buf));
```

```
osip_util_replace_all_lws (tmp);
```

```
[ etc. ]
```

alloca fails and returns NULL

crash!

oSIP version 2.2.0 (December 2004)

```
int osip_message_parse (osip_message_t * sip,  
                        const char *buf, size_t length)
```

```
{ [ ... ]
```

```
char *tmp;
```

```
tmp = osip_malloc (length + 2);
```

```
if (tmp==NULL) { [... print error msg and return -1; ] }
```

```
osip_strncpy (tmp, buf, length);
```

```
osip_util_replace_all_lws (tmp);
```

```
[ etc. ]
```

Related Work

- Static analysis and automatic test generation based on static analysis: limited by symbolic execution technology (see previous discussion)
- Random testing (fuzz tools, etc.): poor coverage
- Dynamic test generation (Korel, Gupta-Mathur-Soffa, etc.)
 - Attempt to exercise a specific program path
 - DART attempts to cover all executable program paths instead (like model checking)
 - Also, DART handles function calls, unknown functions, exploits simultaneous concrete and symbolic executions, is sometimes complete (verification) and has run-time checks to detect incompleteness;
DART has been implemented for C and applied to large examples
- The DART approach (idea, formalization, tool architecture) is independent of specific constraint types or solvers; those params define DART implementations
 - Ex: DART implementation with pointer in-/equality constraints [Sen et al., FSE'05]
- Independent, closely related work on directed search [Cadar-Engler, SPIN'05]

Future Work: Short Term (See IFM'05 Paper)

- Faster constraint solvers
 - Ex: DART on NS with conjunctions only (1) or with disjunctions (2)

depth	error?	Implementation 1	Implementation 2
1	no	5 runs (<1 second)	4 runs (<1 second)
2	no	85 runs (<1 second)	30 runs (<1 second)
3	no	6,260 runs (22 seconds)	554 runs (<1 second)
4	yes	328,459 runs (18 minutes)	9,926 runs (57 seconds)

- More constraint types and decision procedures
 - for pointers, arrays, strings, bit-vectors, etc. (default: random testing)
- Concurrency
 - Scheduling nondeterminism is orthogonal to input data nondeterminism
 - Use partial-order reduction for concurrency (multi-threaded/process)

Future Work: Longer Term (See IFM'05 Paper)

- Combining further static and dynamic software model checking
 - Ex: use program slicing to focus dynamic search towards specific code
 - Ex: use DART as a subroutine to test path feasibility inside static SW MC
- Specifying preconditions (and postconditions)
 - Either using tool-friendly annotations (logic) or input-filtering code
 - How to interpret code as precisely as if specified directly into logic?
 - We need “constraint inference” capabilities...
- Scalability
 - Ex: like static analysis, testing could also be done compositionally
 - When testing $f(g(x))$, $g()$ could be summarized when testing $f()$, using pre/post condition constraints as done for interprocedural static analysis

Conclusions

- Past: two complementary approaches to software model checking
 - Dynamic Approach: Systematic Testing (Ex: VeriSoft)
 - Static Approach: Automatic Abstraction (Ex: SLAM)
- Future: combine both approaches (Ex: DART)
 - DART = Directed Automated Random Testing
 - No manually-generated test driver required (fully automated)
 - As automated as static analysis but with higher precision
 - Starting point for testing process
 - No false alarms but may not terminate
 - Smarter than pure random testing (with directed search)
 - Can work around limitations of symbolic execution technology
 - Symbolic execution is an adjunct to concrete execution
 - Randomization helps where automated reasoning is difficult
- Still plenty of work to do before “software model checking for the masses” !