Software Model Checking via Static and Dynamic Program Analysis

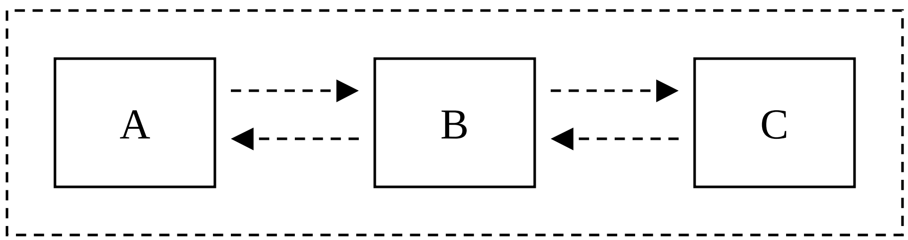
Patrice Godefroid

Bell Laboratories, Lucent Technologies

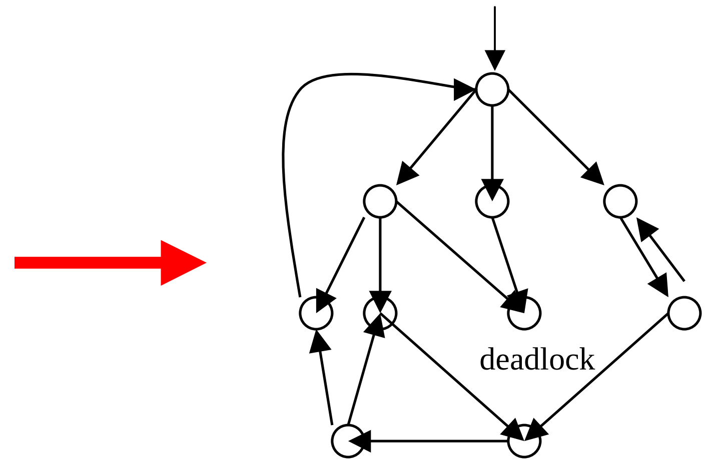
Overview of Software Model Checking

- Part I: The Dynamic Approach (Systematic Testing)
  - VeriSoft
- Part II: The Static Approach (Automatic Abstraction)
  - SLAM and predicate abstraction, 3-valued model checking, generalized model checking
- Part III: Combining the Static and Dynamic Approaches
  - DART, Compositional Dynamic Test Generation (SMART)
- Disclaimer: emphasis on what influenced the speaker, not an exhaustive survey
- Main references: see the bibliography of the abstract

“Model Checking”



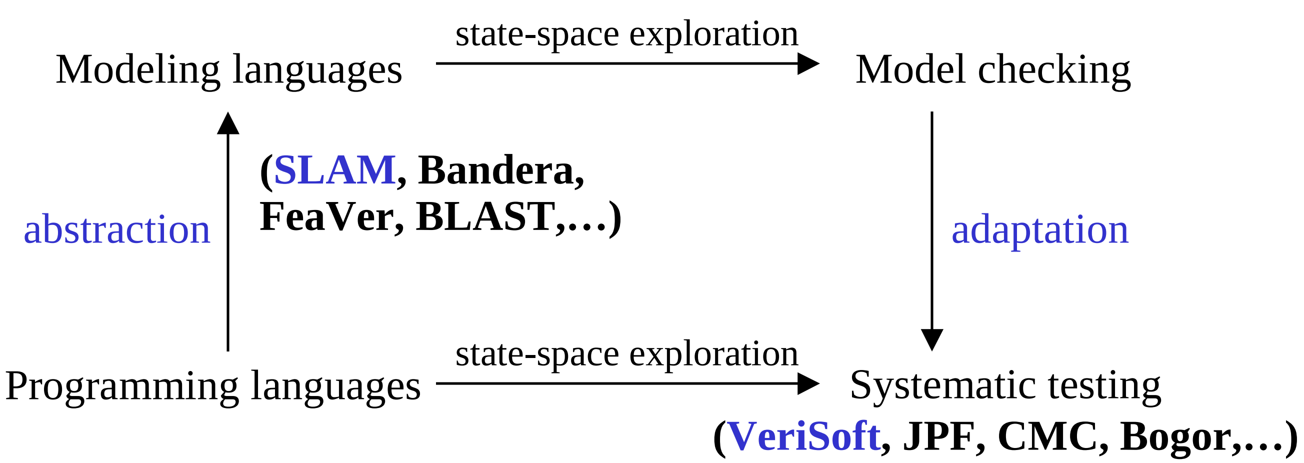
Each component is modeled by a FSM.



- Model Checking (MC) = systematic state-space exploration = exhaustive testing
- “Model Checking” = “check whether the system satisfies a temporal-logic formula”
  - Example: G(p->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:



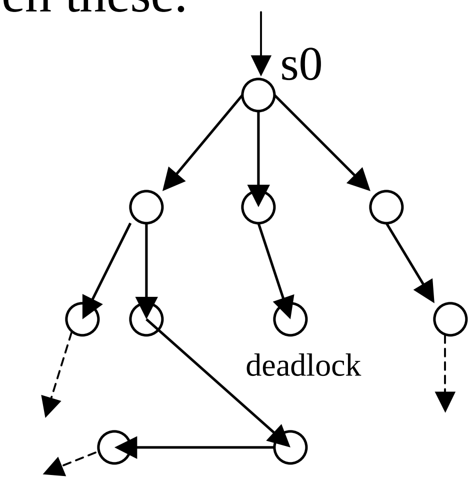
Part I:

The Dynamic Approach (Systematic Testing)

Dynamic Approach: Systematic Testing (VeriSoft)

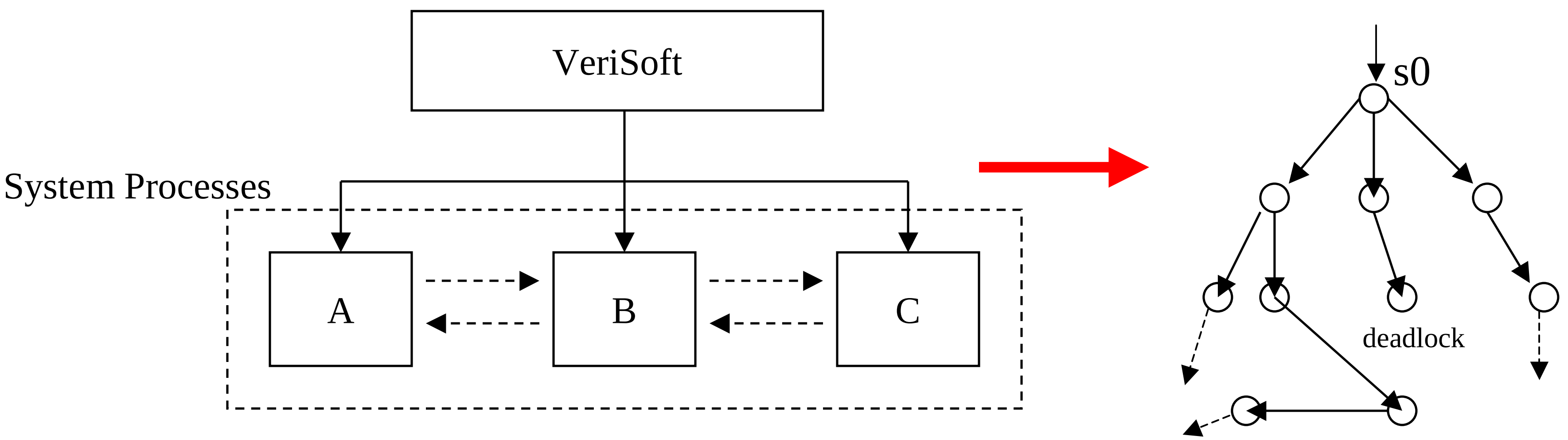
- State Space = "product of (OS) processes" (Dynamic Semantics)
  - Processes communicate by executing operations on com. objects.
  - Operations on com. objects are visible, other operations are invisible.
  - Only executions of visible operations may be blocking.
  - The system is in a global state 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

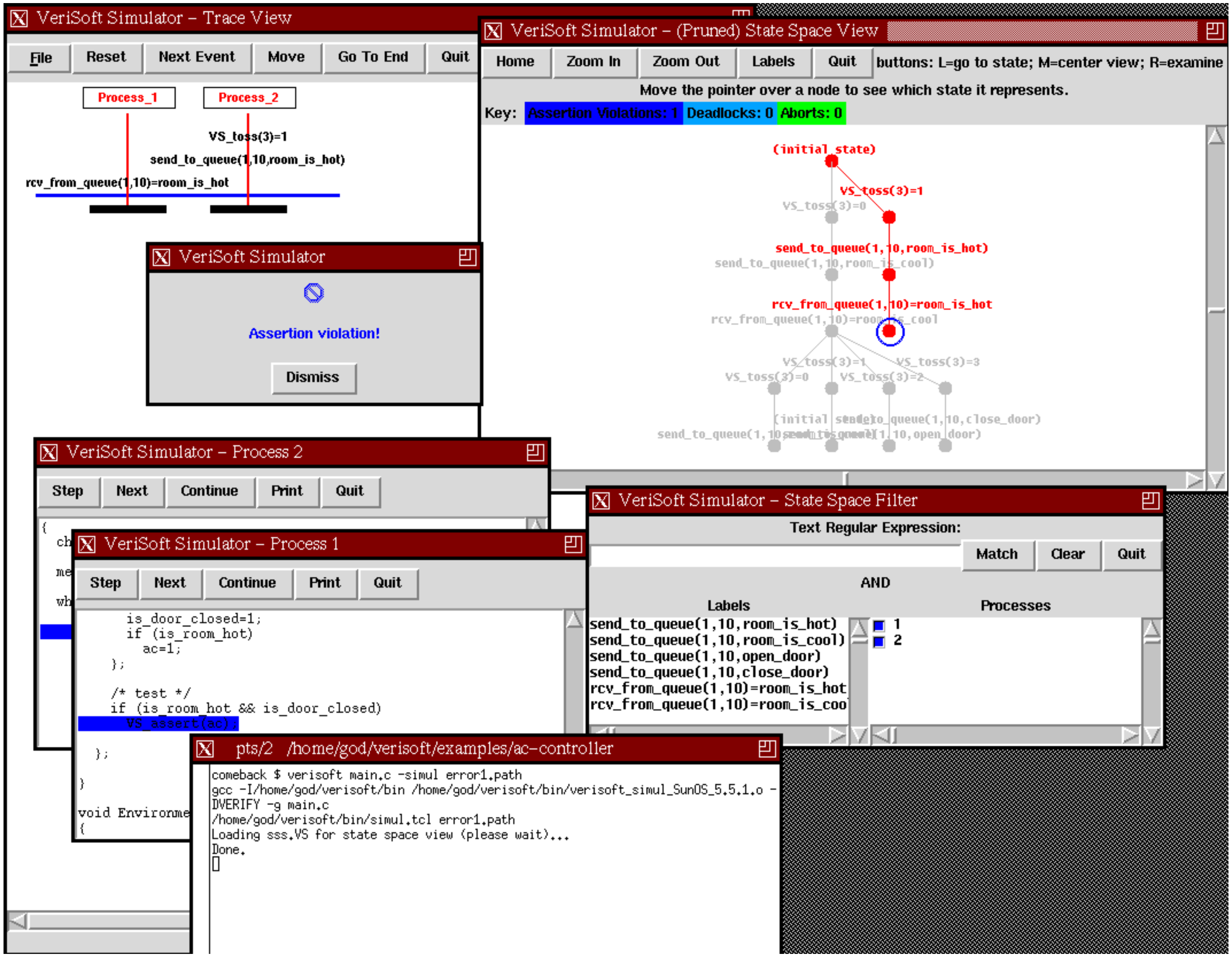
- Controls and observes the execution of concurrent processes of the system under test by intercepting system calls (communication, assertion violations, etc.).
- Systematically drives the system along all the paths (=scenarios) in its state space (=automatically generate, execute and evaluate many scenarios).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



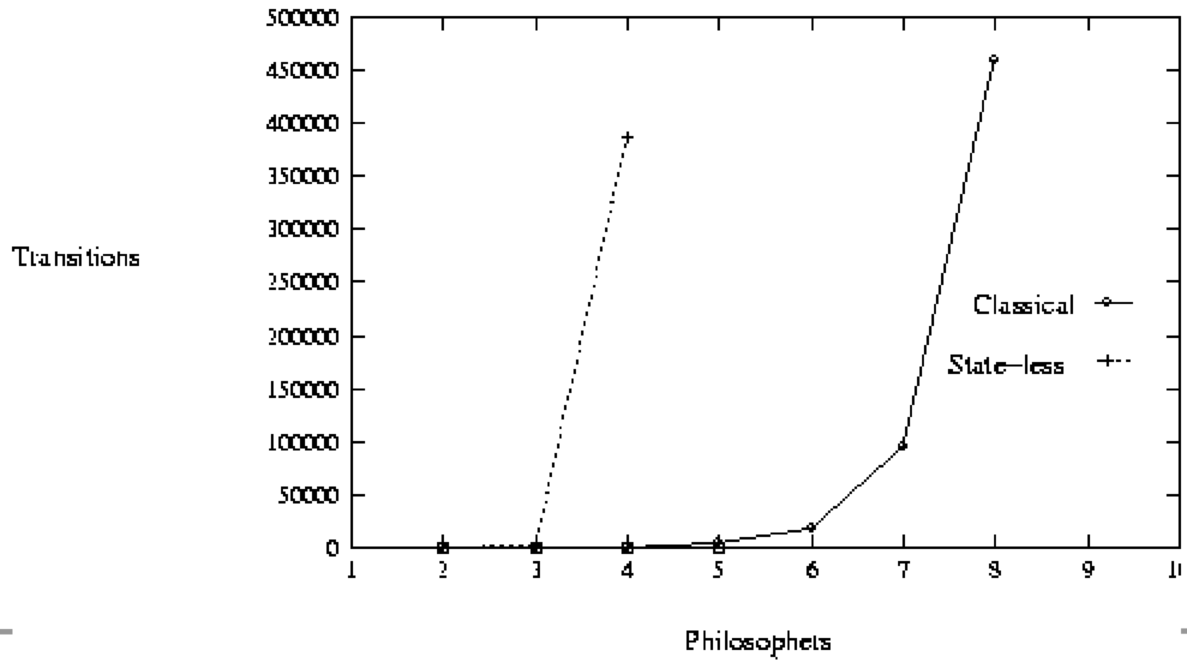
Originality of VeriSoft

- VeriSoft is the 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 <u>models</u> of software systems;
  - each state is represented by a <u>unique identifier</u>;
  - visited states are saved in memory (hash-table, BDD,…).
- With <u>programming languages</u>, 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!

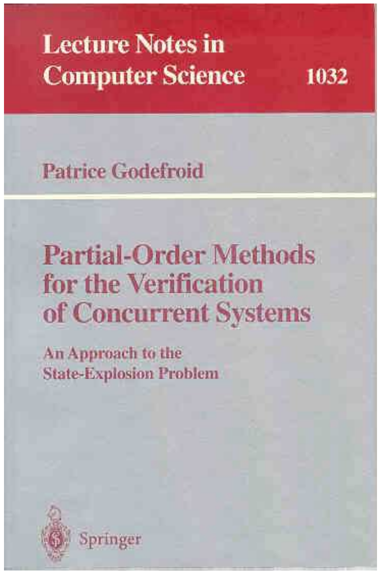


Partial-Order Reduction in Model Checking

- 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,…)



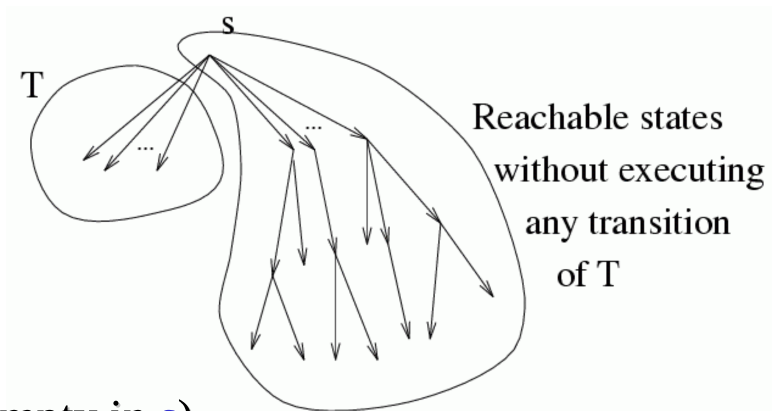
[Note: checking more elaborate properties require other extensions

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

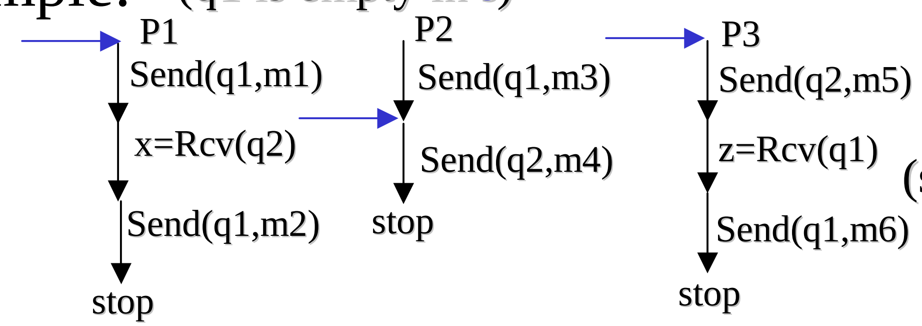
Not used here as VeriSoft only checks reachability properties]

Persistent/Stubborn Sets

- Intuitively, a set T of enabled transitions in s are <u>persistent in s</u> 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

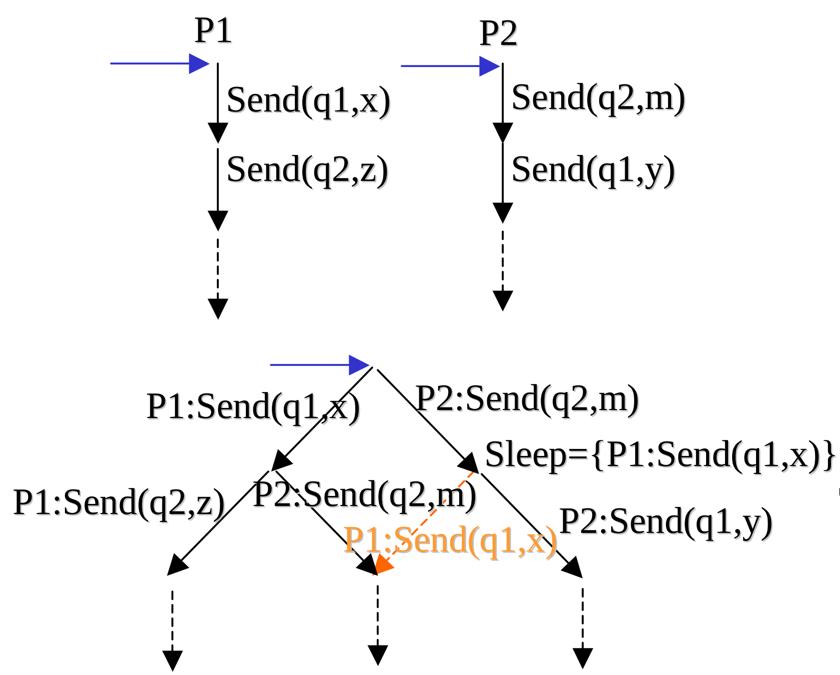
The most advanced algorithms for (statically) computing persistent sets are based on "stubborn sets" [Valmari]

- 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:

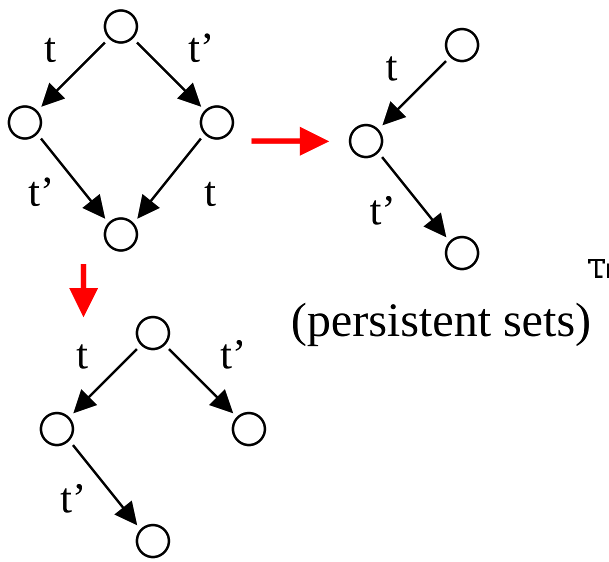


Transitions in Sleep are not explored!

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

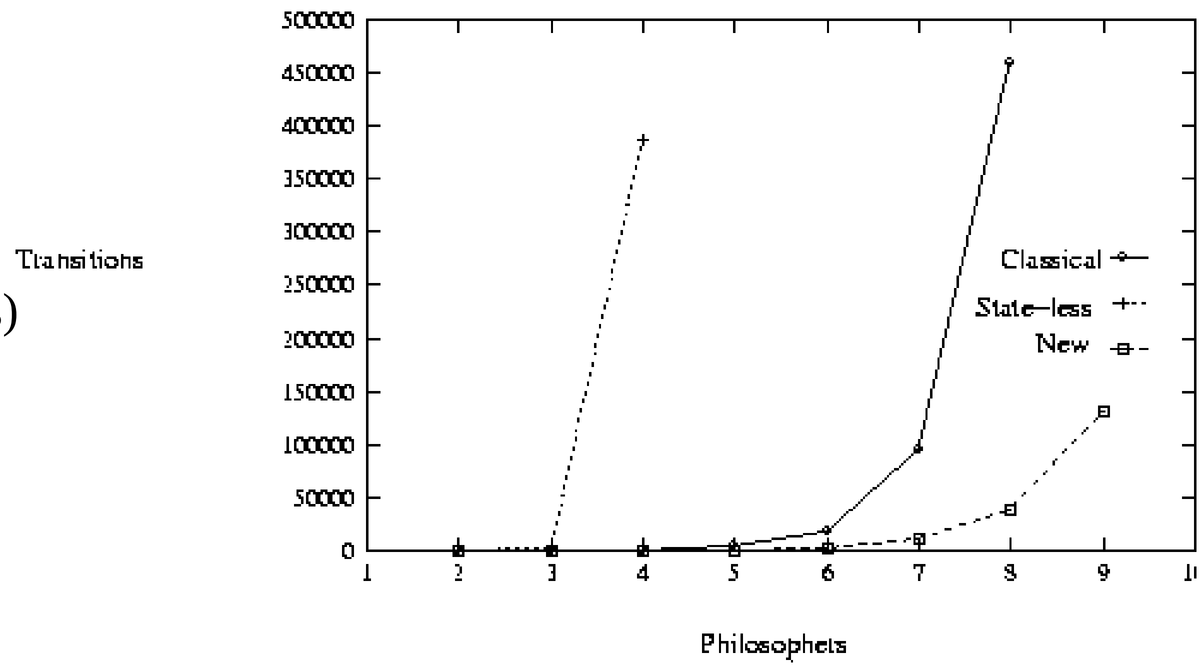
An Efficient State-Less Search

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



(persistent sets)

(sleep sets)



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

VeriSoft - Summary

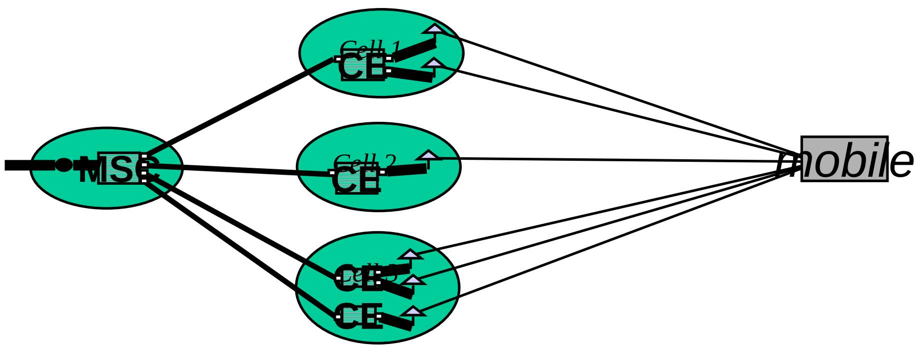
- Two key features distinguish VeriSoft from other model checkers
  - Does not require the use of any specific modeling/programming language.
  - Performs a state-less search.
- Use of partial-order reduction is key in presence of concurrency.
- In practice, the search is typically incomplete.
- From a given initial state, VeriSoft can always guarantee a <u>complete coverage</u> of the state space <u>up to some depth</u>.

Users and Applications

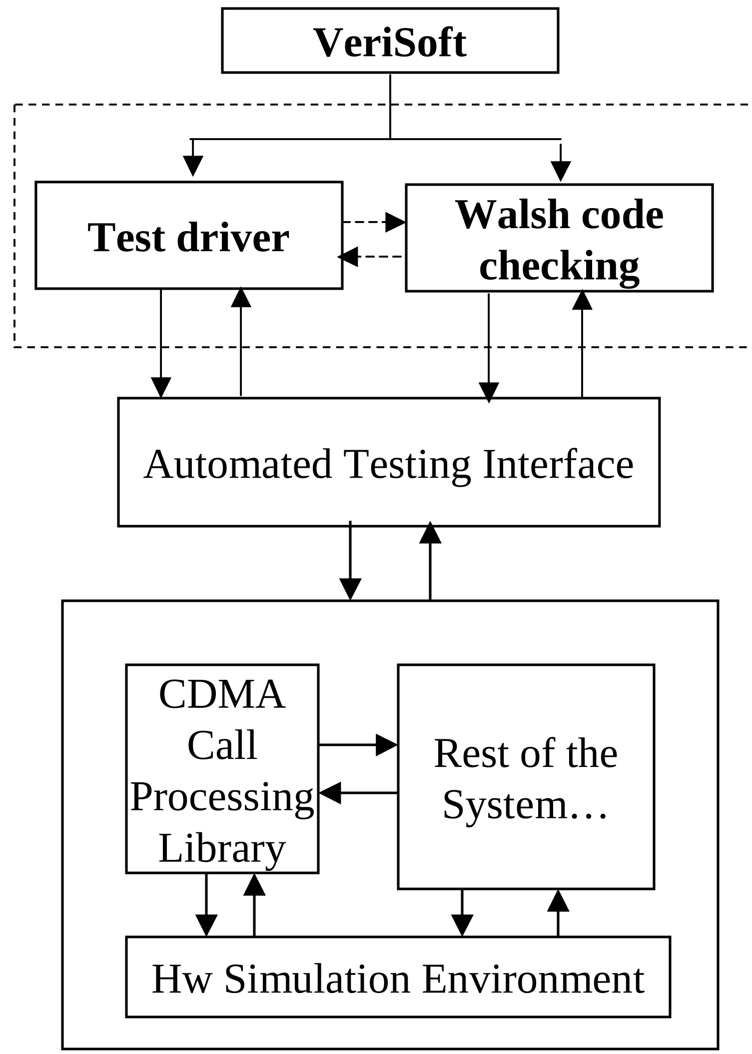
- Development of research prototype started in 1996.
- VeriSoft 2.0 available outside Lucent since January 1999:
  - 100's of licenses in 25+ countries, in industry and academia
  - Free download at http://www.bell-labs.com/projects/verisoft
- Examples of applications in Lucent:
  - 4ESS HBM unit testing 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)

Example of Industrial Application: CDMA

- 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?”

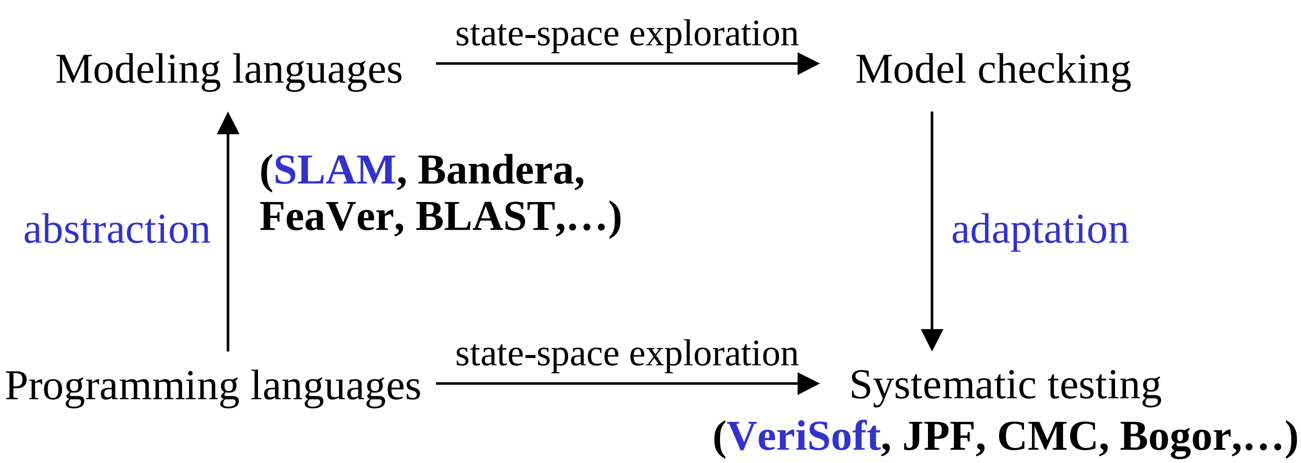
Part II:

The Static Approach (Automatic Abstraction)

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:



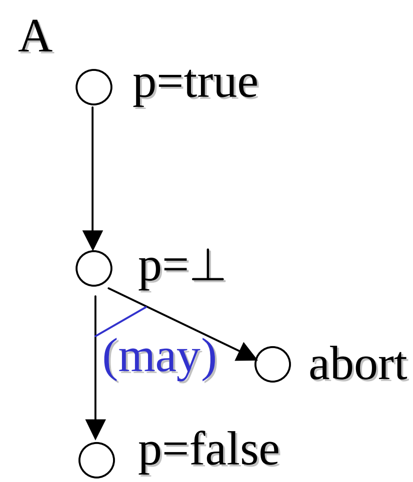
Static Approach: Automatic Abstraction (SLAM)

"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 $\longrightarrow$ 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,Sutre,…], and many more!

Abstraction for Verification ***and*** Falsification

Using 3-valued models and logics, Generalized Model Checking…

See other slides here:

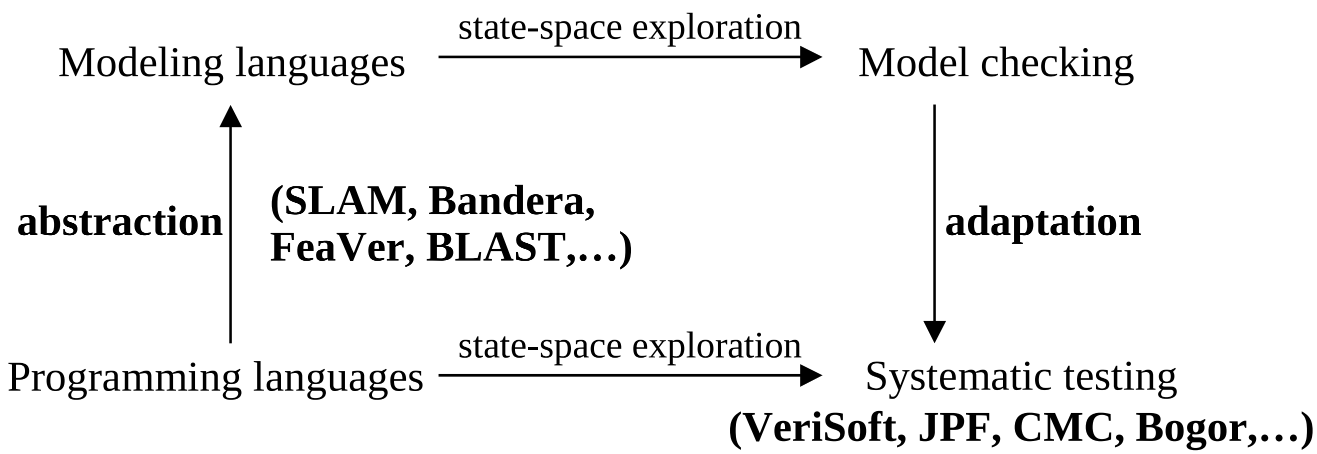
Slides.pdf

Part III:

Combining the Static and Dynamic Approaches

Model Checking of Software: Today

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)

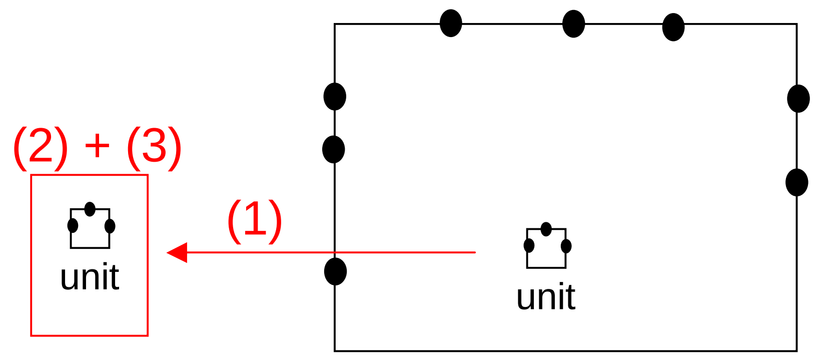
Model Checking of Sofware: What Next?

- General idea: combine static and dynamic analysis
- 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] with N. Klarlund and K. Sen (summer intern, 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
  - One way to combine static and dynamic analysis for SW 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



Any program (that compiles) can be run and tested automatically:

No need to write any test driver or harness code!

DART detects program crashes, assertion violations, etc.

Example (C code)

```
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 */
  }
}
```

(1) Interface extraction:
- parameters of top-level function
- external variables
- return values of external functions

(2) Generation of test driver for random testing:

```
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

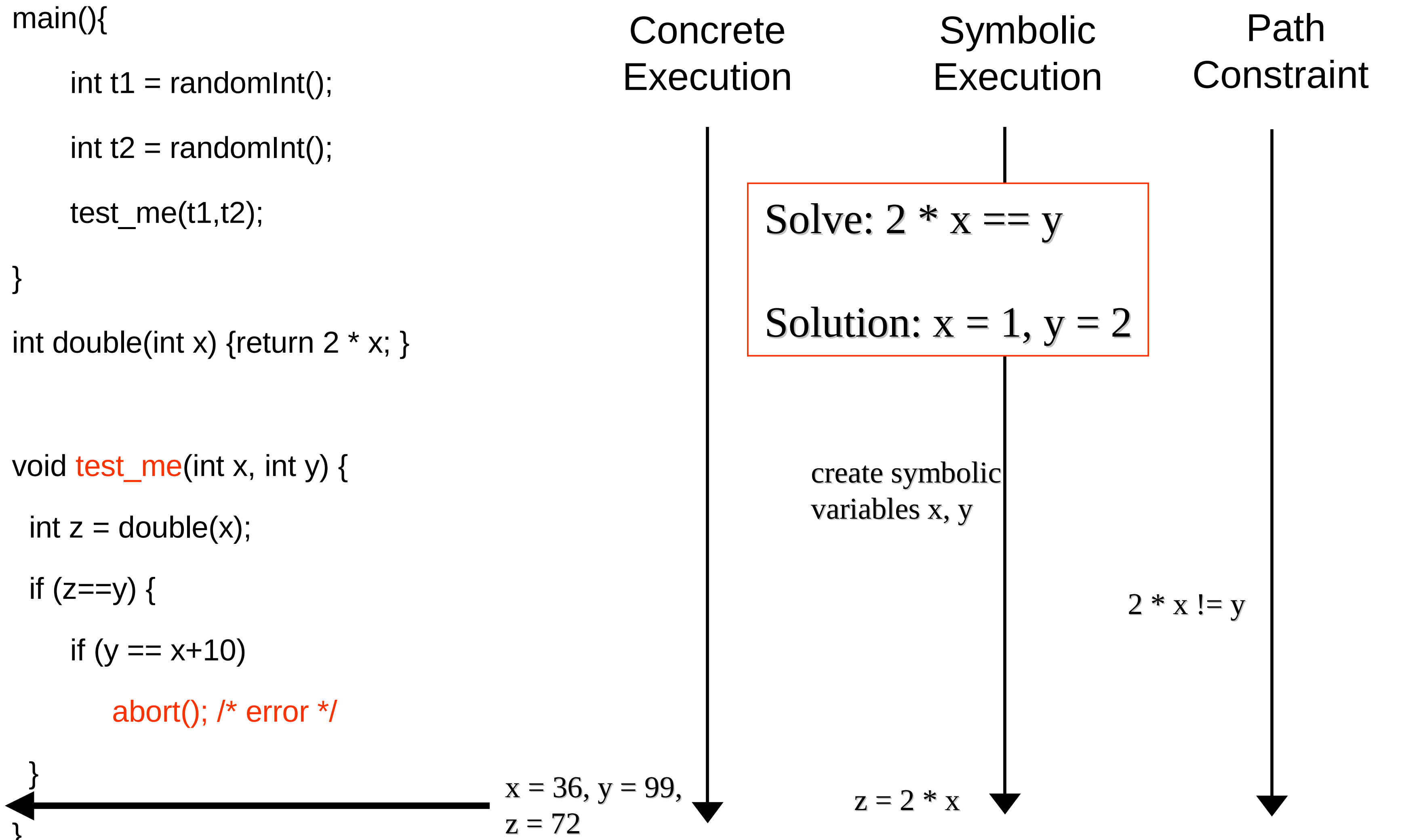
DART Step (3): Directed Search

```
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 */
  }
}
```



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	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

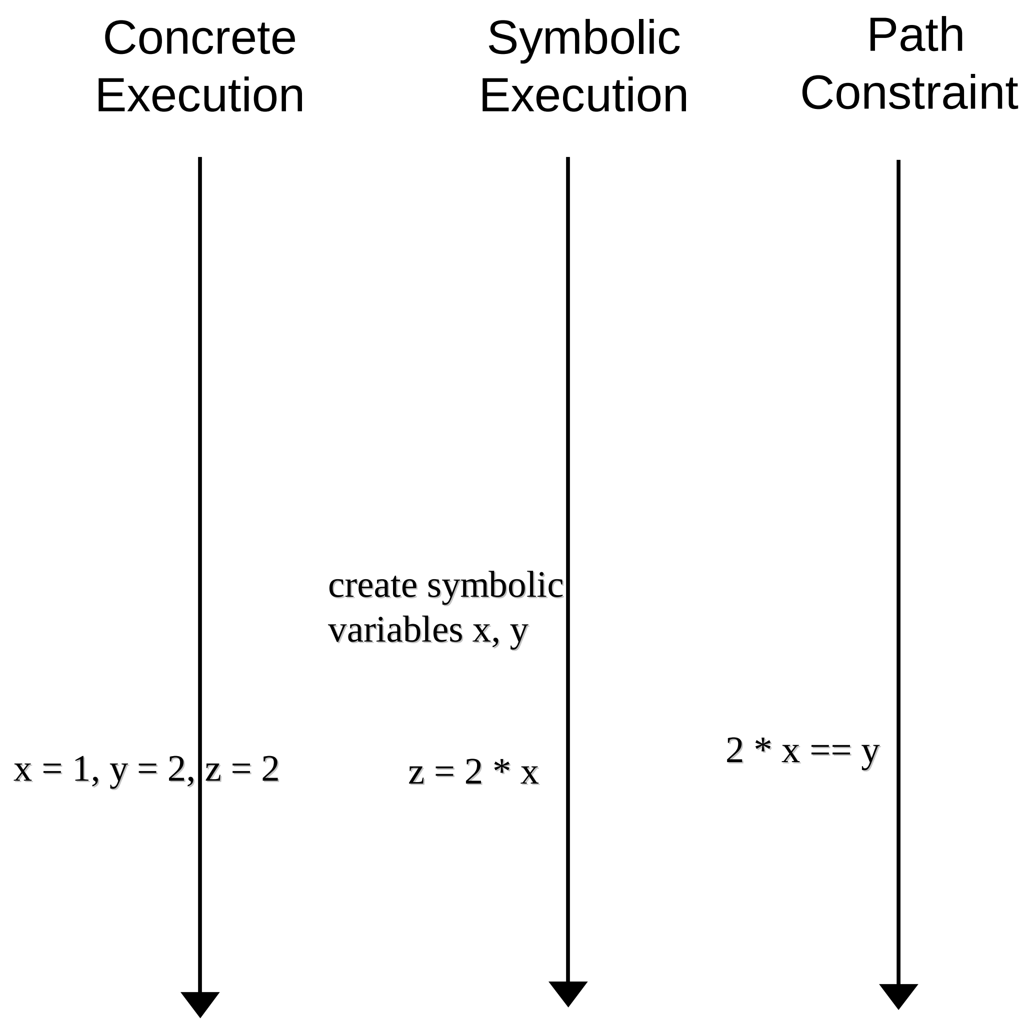
create symbolic variables x, y

x = 1, y = 2, z = 2

z = 2 * x

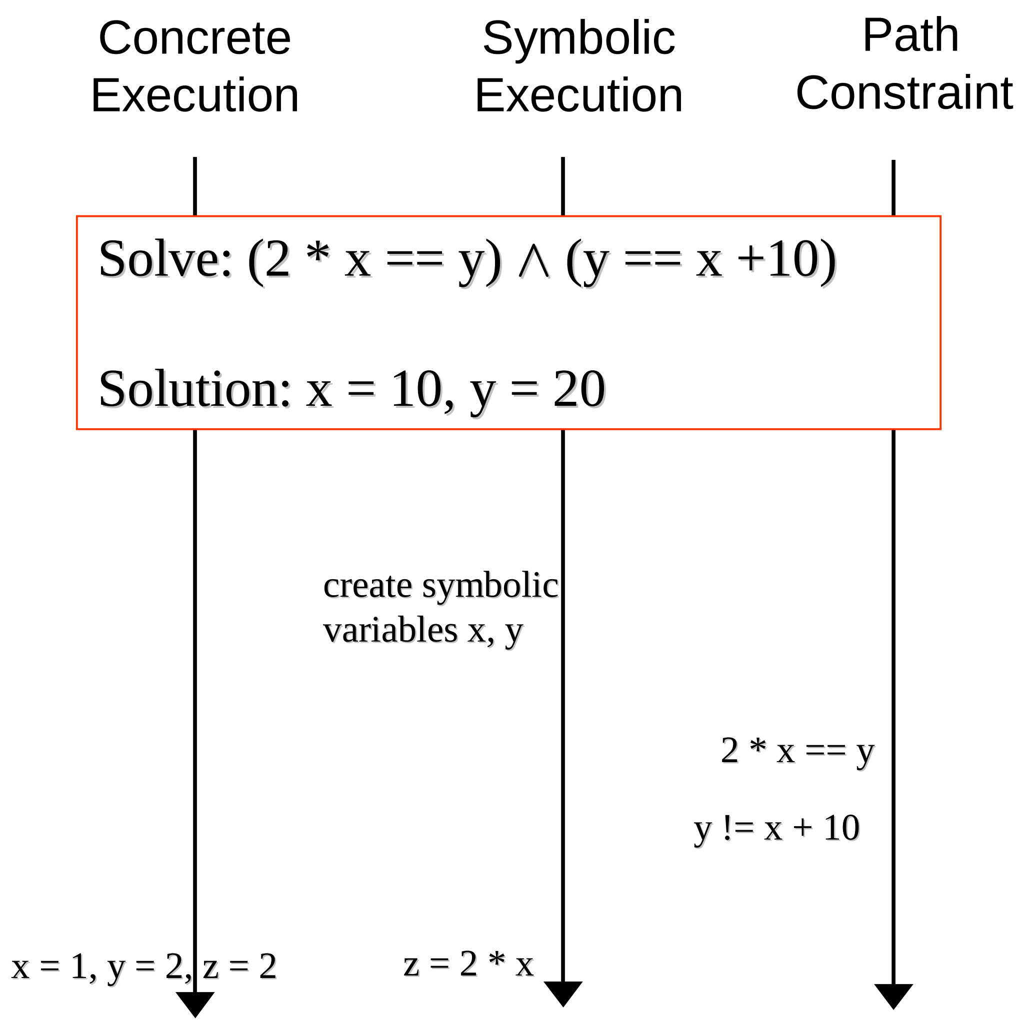
DART Step (3): Directed Search

```
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 */
  }
}
```



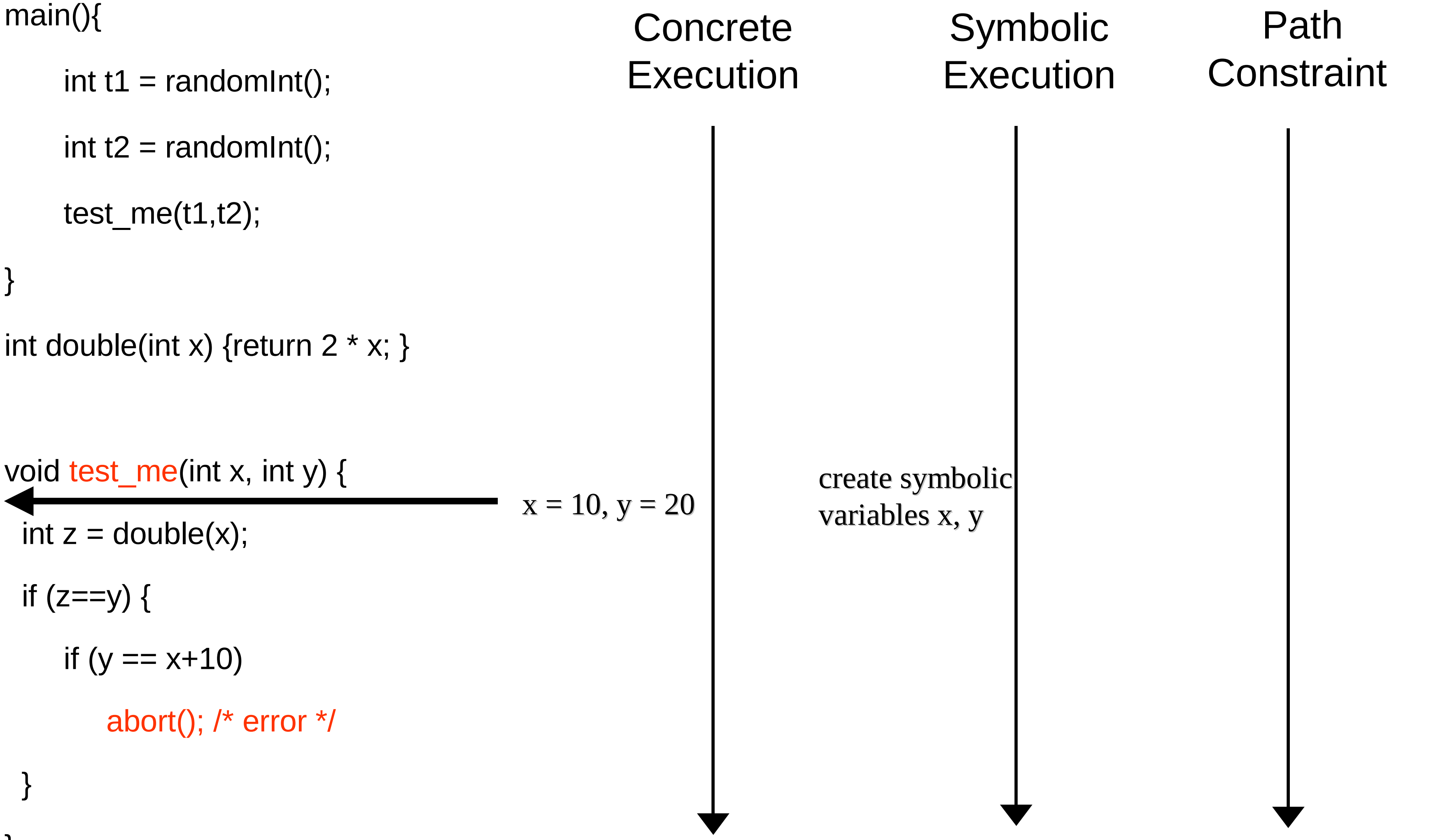
DART Step (3): Directed Search

```
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 */
  }
}
```



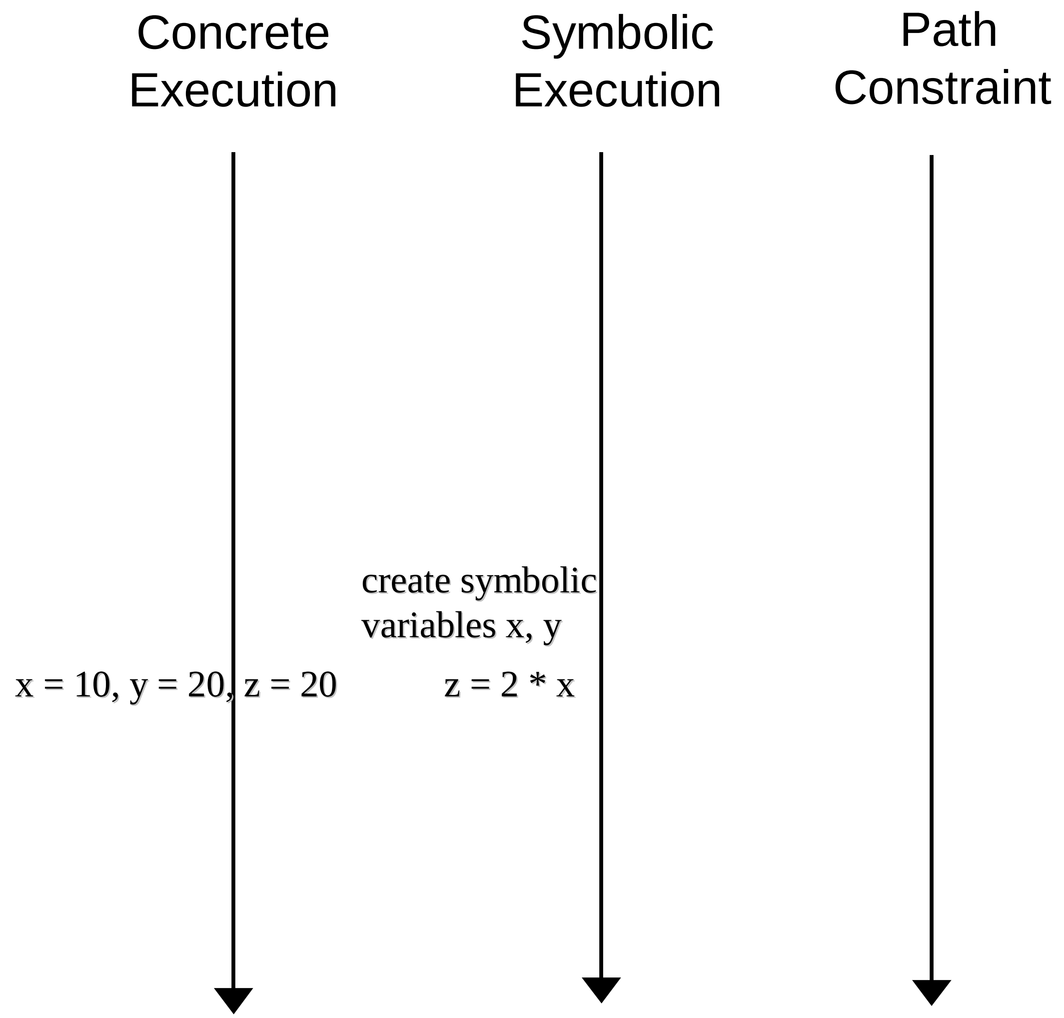
DART Step (3): Directed Search

```
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 */
  }
}
```



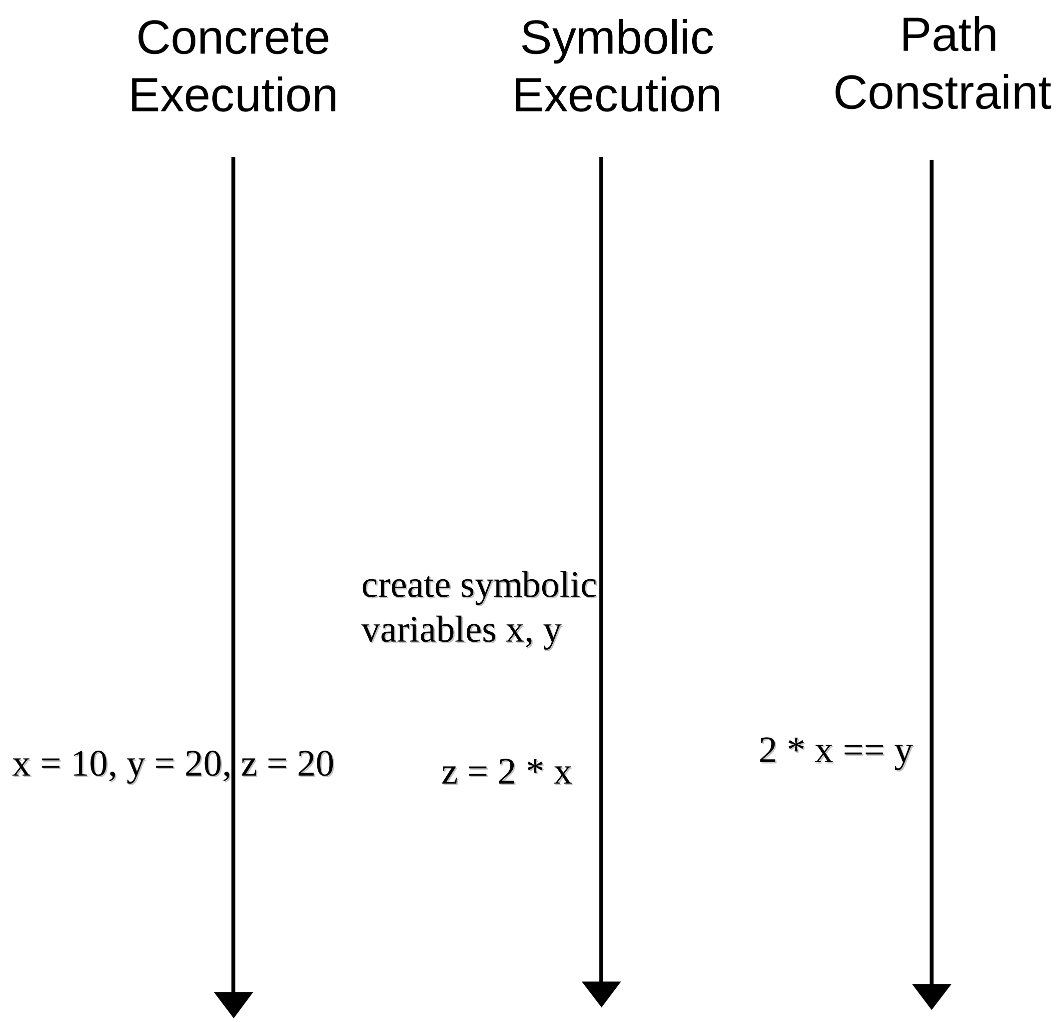
DART Step (3): Directed Search

```
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 */
  }
}
```



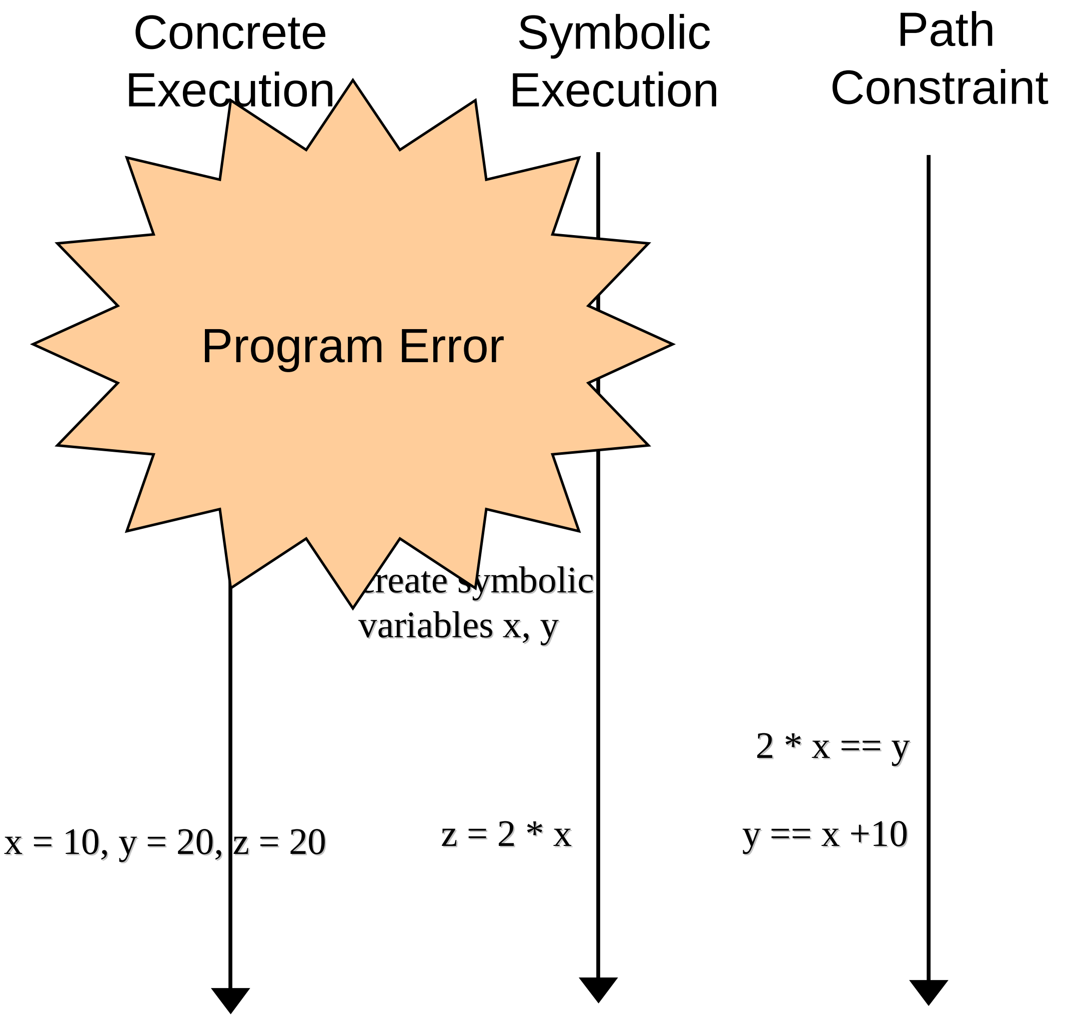
DART Step (3): Directed Search

```
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 */
  }
}
```



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: Use of Concrete Values in Symbolic Execution

```
void foo(int x,int y){
    int z = x*x*x; /* could be z = h(x) */
    if (y == z) {
        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: Use of Concrete Values in Symbolic Execution

```
void foo(int x,int y){
    int z = x*x*x; /* could be z = h(x) */
    if (y == z) {
        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 y != 27
- Solve y == 27 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 }
```

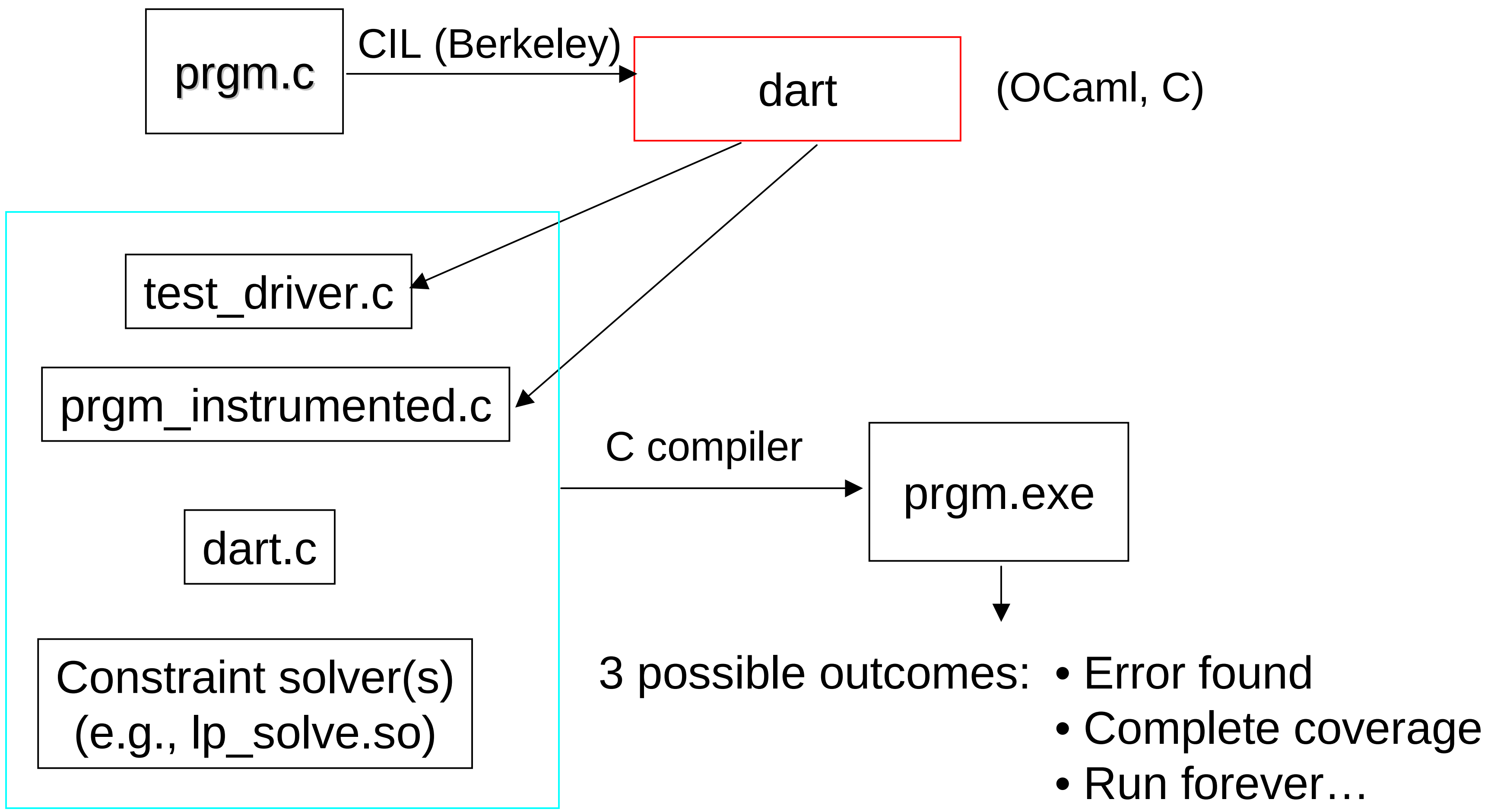
- Symbolic execution is stuck at line 2…
- Static analysis tools will conclude that both aborts may be reachable
  - "Sound" tools will report both, and thus one false alarm
  - "Unsound" tools will report "no bug found", and miss a bug
- Static-analysis-based test generation techniques are helpless here !!!
- In contrast, DART finds the only error (line 4) with high probability (but cannot prove line 8 is unreachable)
- Unlike static analysis, all bugs reported by DART are guaranteed to be sound

Other Advantages of Dynamic Analysis

```
struct foo { int i; char c; }

bar (struct foo *a) {
    if (a->c == 0) {
        *((char *)a + sizeof(int)) = 1;
        if (a->c != 0) {
            abort();
        }
    }
}
```

- 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



Some Experimental Results

Experimental results with a DART prototype for C are very encouraging:

- Benchmark: Needham-Schroeder authentication protocol (400 lines of C code with a known attack)
  - DART takes about 1 min (9,926 runs) to discover the known attack (1GHz P-III)
  - Previous tools (like VeriSoft, BLAST, static analyzers,…) do not find the attack
    - VeriSoft does not find the attack in 24 hours of search (albeit with a different, concurrent and nondeterministic, Dolev-Yao intruder model)
    - BLAST reports a spurious error after 6 minutes of search (due to imprecision of current alias-analysis used), or hangs with “interpolant” optimization turned on (after a call to Simplify with a formula containing 40,000+ variables and 68,000+ clauses)

- oSIP (Open Source SIP library; 30,000 lines of C code)
  - DART found a way to crash 65% of the 600 externally visible functions in the oSIP API within 1,000 runs per function
  - Analysis revealed a new attack to crash the oSIP parser (by remotely send it a single particular message!)

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 has been implemented for C and applied to large examples (handles full C, function calls, unknown functions, exploits simultaneous concrete and symbolic executions, has run-time checks to detect incompleteness,…)
- Independent, closely related work on directed search [Cadar-Engler, SPIN'05]
- 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]
  – Ex: DART implementation with bit-level symbolic execution [Engler et al., S&P'06]

New Results: Introducing SMART (to appear)

- Problem: Executing all feasible program paths does not scale!
  - Number of paths can be exponential (even if loop-free) or infinite (loops)
  - E.g., in oSIP, branch coverage stuck around 30% due to path explosion…
- Idea: compositional dynamic test generation (SMART algorithm)
  - Like interprocedural static analysis: use summaries of individual functions
  - If f() calls g(), analyze/test g() separately, summarize the results, and use g()'s summaries when testing f()
    - summaries may now include information about concrete values
    - g()'s outputs are treated as symbolic inputs to f()
  - Strategies for computing summaries:
    - bottom-up: easier to implement but many unused summaries
    - top-down: compute summaries on a demand-driven basis

SMART = "Systematic Modular Automated Random Testing"

SMART = Modular DART

Theorem: SMART provides same path coverage as DART

- Same “local path” reachability, branch coverage, assertion violations,…

```
// locate index of first character c in s
int locate(char *s, int c) {
  int i=0;

  while (s[i] != c) {
    if (s[i] == 0) return -1;
    i++;
  }
  return i;
}
void top(char *input) {
  int z;

  z = locate(input,'a');
  if (z == -1) return -1;            // error
  if (input[z+1] != ':') return 1; // success
  return 0;                          // failure
}
```

- Assume input (and s) are null-terminated and of maximum length n
- locate() has at most 2n execution paths
  Ex of summaries:
  (s[0] == c) => ret = 0
  (s[0] != c) & (s[0] == 0) => ret = -1
  (s[0] != c) & (s[0] != 0) & (s[1] == c) => ret = 1
  etc.
- top() has at most 3 execution paths
- P={top(),locate()} has at most 3n execution paths
- DART search algorithm explores 3n paths
- SMART search algorithm explores 2n+2paths
  Sum vs. product: linear vs. exponential!
  (Similar to HSM/PDS verification…)
- Claim: SMART search is necessary to make the “DART approach” scalable!

Extensions (see [IFM'2005])

- 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’2005])

- 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 analyzer
- Specifying preconditions (and postconditions)
  - Either using tool-friendly annotations (logic) or input-filtering code
  - How to interpret code as precisely as if specified directly in logic?
    We need “constraint inference” capabilities…

```
int locate(char *s, int c) {
 int i=0;

 while (s[i] != c) {
   if (s[i] == 0) return -1;
   i++;
 }
 return i;
}
```

From

$(s[0] == c) \Rightarrow ret = 0$

$(s[0] != c) \,\&\, (s[0] == 0) \Rightarrow ret = -1$

$(s[0] != c) \,\&\, (s[0] != 0) \,\&\, (s[1] == c) \Rightarrow ret = 1$

etc.

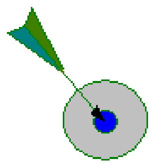
To

$\exists\, i : s[i] == c \,\&\, (\forall\, j<i: (s[j] != c) \,\&\, (s[j] != 0)) \Rightarrow ret=i$

etc.

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” !