Symstra: A Framework for Generating Object-Oriented Unit Tests using Symbolic Execution

Software Testing and Validation

Group 18
João Santos 110947 | Diogo Pereira 110996

04 de junho, 2024

Motivation for Symstra: Enhancing Unit Test Generation

Importance of Unit Testing:

- Ensures Code Quality
- Early Error Detection

Challenges in Manual Test Generation:

- Time-Consuming
- Prone to Errors and Omissions

Limitations of Existing Automated Tools:

- Random Test Generators:
  - Jtest and JCrasher, often result in repetitive sequences and incomplete coverage.
- Model-Based Testing:
  - Tools like AsmLT require large concrete domains and precise abstraction functions.
  - They struggle with complex data structures and state space explosion.

Structure of the Talk

- **Introduction**
- **Proposed Solution: Symstra**
  - Symstra Framework Overview
  - Symbolic Execution
  - State subsumption
  - Heap Isomorphism
  - Symbolic State Exploration
  - Concrete Test Generation
- **Evaluation**
- **Conclusions & Future Directions**
- **Q&A**

Introduction: Context and Technical Problem

Context

- Need for effective and automated unit test generation in object-oriented programming.

Technical Problem:

- Creating sequences of method invocations that adequately test object interactions is challenging.

Why It’s Hard:

- Manual creation is labor-intensive and error-prone.
- Existing tools lack comprehensive coverage, especially for complex data structures.

Proposed Solution: Symstra

Symstra Framework:

- Uses **symbolic execution** to:
  - exhaustively explore method sequences of the CUT;
  - operate with symbolic arguments that represent multiple concrete values.
- Test Generation:
  - Generates unit tests that cover various object states and method interactions.

Goals:

- Achieve higher branch coverage.
- Reduces the time required to generate comprehensive unit tests.
- Improves the efficiency of the test generation process through **state subsumption** and **heap isomorphism**.

Example: Binary Search Tree (BST)

Example: **Binary Search Tree (BST)**

- Contains standard set operations: insert, remove, and contains.
- Some tools such as Jtest or JCrasher test a class by generating random sequences of methods; for BST, they could for example generate the following tests:

```
Test 1:
  BST t1 = new BST();
  t1.insert(0);
  t1.insert(-1);
  t1.remove(0);
```

```
Test 2:
  BST t2 = new BST();
  t2.insert(2147483647);
  t2.remove(2147483647);
  t2.insert(-2147483648);
```

- **Symstra** also explores all sequences, but using **symbolic values** for primitive-type arguments in method calls.

Symbolic **Execution**

```
BST t = new BST();
t.insert(x₁);
t.insert(x₂);
t.insert(x₃);
t.remove(x₄);
```

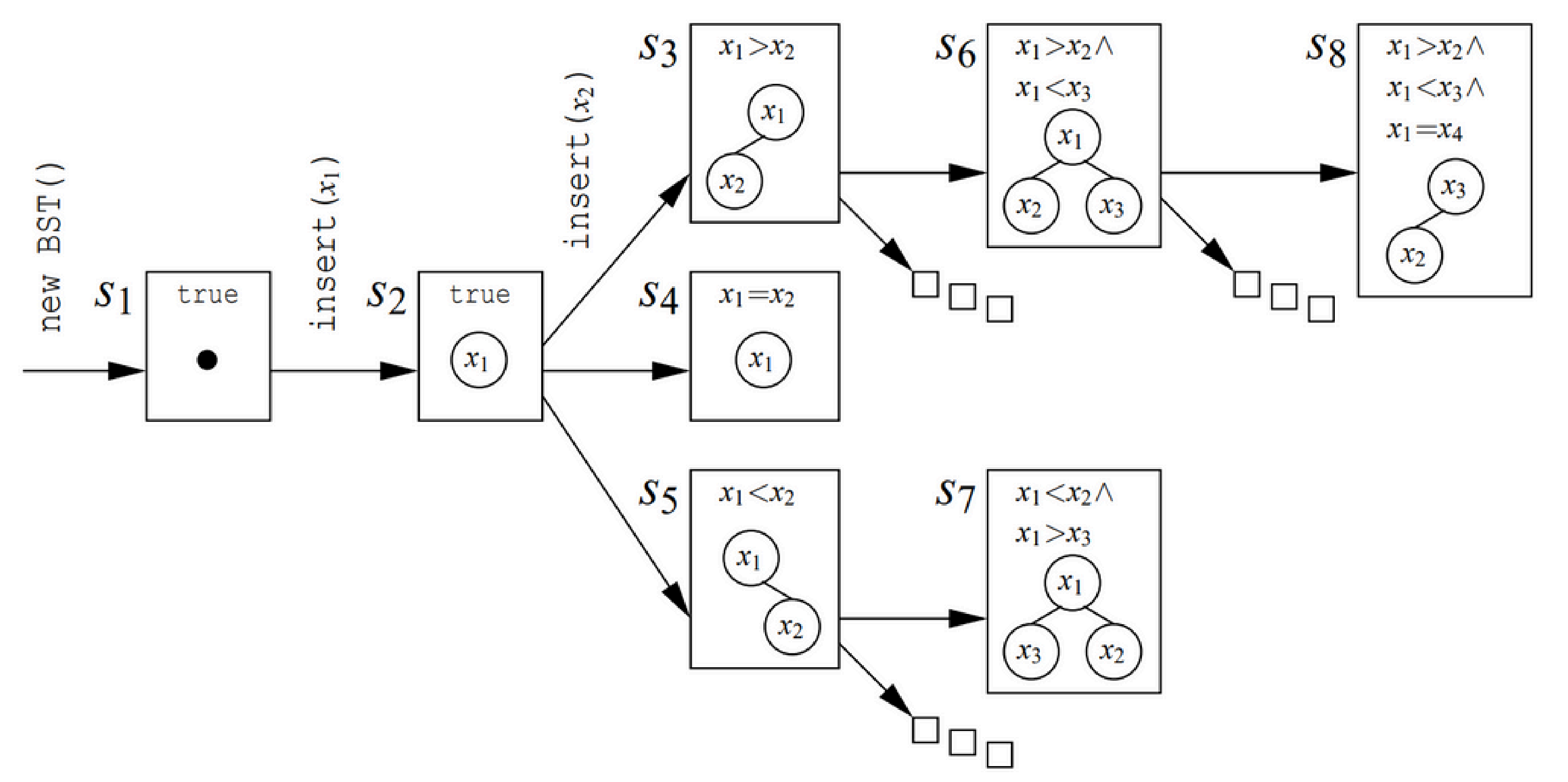
x1, x2, x3, x4 are symbolic variables

- Having **symbolic arguments** necessitates **symbolic execution**.
- It operates on a **symbolic state** that consists of two parts:
  - a **constraint (path condition)**, that must hold for the execution to reach a certain point;
  - a **heap** that contains symbolic variables.

- While an execution of a sequence with concrete arguments produces one state, **symbolic execution of a sequence with symbolic arguments can produce several states**, thus resulting in an **execution tree**.

Execution **Tree**

```
BST t = new BST();
t.insert(x1);
t.insert(x2);
t.insert(x3);
t.remove(x4);
```

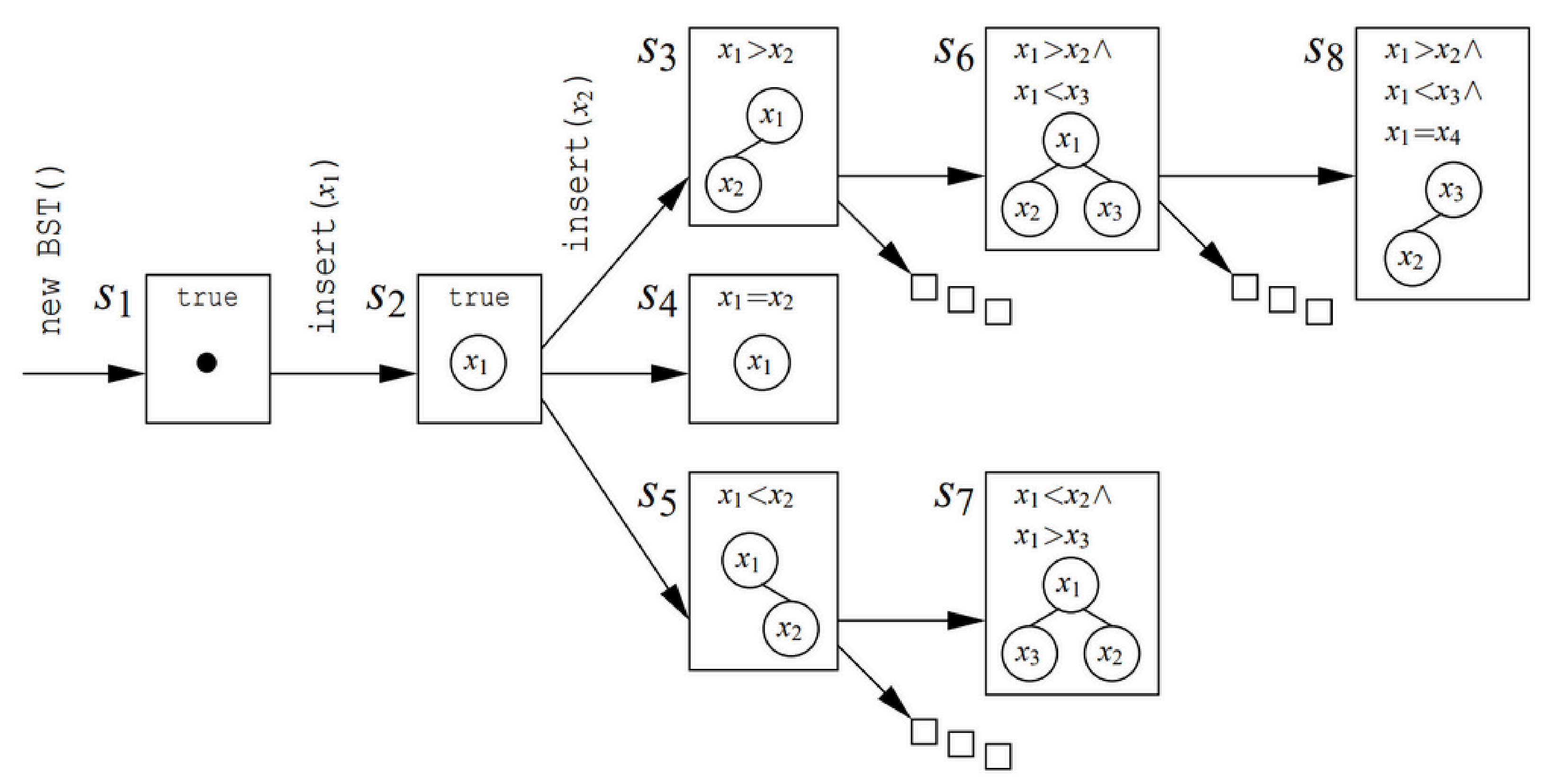


- Symbolic states **s2** and **s4** are *syntactically different*:
  - **s2** has the constraint ***true***; while **s4** has ***x1 = x2***
- However, these two **symbolic states** are *semantically equivalent*:
  - They can **produce the same set of concrete heaps** by giving to *x1* and *x2* concrete values that satisfy the constraints;
  - since *x2* does not appear in the heap in *s4*, the constraint in *s4* is "irrelevant".

Instead of state equivalence, it suffices to check *state subsumption*

State subsumption

```
BST t = new BST();
t.insert(x1);
t.insert(x2);
t.insert(x3);
t.remove(x4);
```



- ***s2* subsumes *s4*** because the set of concrete heaps of ***s4*** is a **subset** of the set of concrete heaps of ***s2***.

Hence, Symstra **does not need to explore *s4* after it has already explored *s2*.**

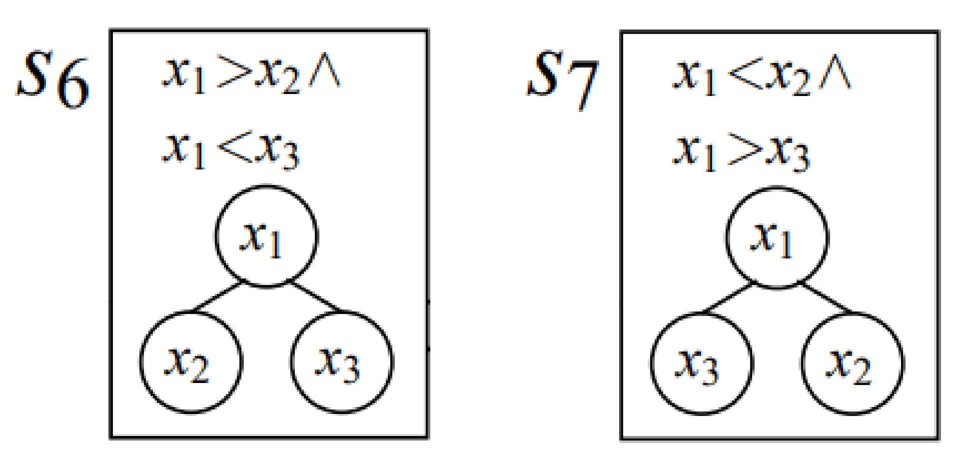
- It detects this by checking that the implication of constraints ***x1 = x2* ⇒ *true* holds**.

- Now, ***s6*** & ***s7*** are ***syntactically different*, but *semantically equivalent***: we can exchange the variables *x2* and *x3* to obtain the **same symbolic state**. Symstra detects this by checking that *s6* and *s7* are **isomorphic**.

Heap **Isomorphism**

- **Heap Isomorphism**: Two heaps are **isomorphic** if they can be transformed into each other by renaming nodes/symbolic variables, preserving the structure.

- **Importance:** Detecting isomorphic heaps helps in **reducing redundant state exploration** by identifying equivalent method behaviors.



we can exchange the variables x2 and x3 to obtain the same symbolic state.

- **Isomorphism Detection:** Symstra linearizes heaps into integer sequences to check for isomorphism efficiently. This involves:
  - Each object and symbolic variable in the heap is assigned with a unique identifier.
  - Traversing the heap depth-first to create a sequence representation.
  - Comparing these sequences to determine **isomorphism**.

Symbolic State Exploration

- The state space consists of **all states reachable through symbolic execution** of all possible method sequences for the CUT.
- State Exploration Process:
  - Symstra explores the symbolic state space using **Breadth-First Search**.
    - Inputs: Set of constructors C, methods M, and a bound on sequence length.
  - Use a **queue** to manage states for breadth-first search
  - For each state, **symbolically execute each method** (while the queue is not empty, pop the first state)
    - Each execution path produces a new symbolic state.
  - **Add new states to the queue** only if they are not subsumed by already explored states.
  - Continue the process until the queue is empty, ensuring that all possible method sequences up to a given length are explored.

Final Step: Concrete Test Generation

- <u>During symbolic state exploration</u>, Symstra **generates specific concrete tests** for the explored states.
- Each symbolic state ⟨C, H⟩ has an associated method sequence and constraint C
- **Process**:

1. Based on the state ⟨C, H⟩, Symstra tracks the shortest method sequence leading to that state.
2. Using the POOC solver, Symstra solves the constraint C to find concrete values for the symbolic variables.
3. These method sequences are then converted into JUnit tests.

Summary of Symstra Execution Flow

1. **Initialization:**
   - Begin with an **initial symbolic state** $\langle$***true, {}***$\rangle$.

2. **Symbolic Execution:**
   - Execute each method symbolically using symbolic arguments.
   - Explore both branches of conditional statements.
   - Update path conditions with branch conditions or their negations.

3. **Create Symbolic States:**
   - Generate new symbolic states after each method invocation.

4. **State Comparison:**
   - <u>Subsumption</u>:
     - Check if a new state is subsumed by an already explored state.
     - If subsumed, prune the exploration of the new state.
   - <u>Isomorphism:</u>
     - Detect isomorphic states to avoid redundant exploration.
     - Use linearization to efficiently compare heaps.

5. **Concrete Test Generation:**
   - Convert symbolic states into concrete test cases.
   - Use constraint solvers to find concrete values that satisfy the constraints.

Evaluation: Symstra & Rostra

- **Symstra** was developed on top of **Rostra**, a previous framework
- **Metrics** to compare **Symstra vs Rostra**:
  - **Time to Generate Tests:** Measures the efficiency of Symstra in producing test cases
  - **Number of States Explored**: Indicates the thoroughness of state space exploration
  - **Number of Tests Generated:** Indicates the number of tests generated
  - **Branch Coverage Achieved:** Reflects the effectiveness in covering different code paths

Evaluation:
Symstra & Rostra

- **Classes** and **methods** used to compare **Symstra vs Rostra**:

class	methods under test	some private methods	#ncnb lines	# branches
IntStack	push,pop	–	30	9
UBStack	push,pop	–	59	13
BinSearchTree	insert,remove	removeNode	91	34
BinomialHeap	insert,extractMin delete	findMin,merge unionNodes,decrease	309	70
LinkedList	add,remove,removeLast	addBefore	253	12
TreeMap	put,remove	fixAfterIns fixAfterDel,delEntry	370	170
HeapArray	insert,extractMax	heapifyUp,heapifyDown	71	29

Table 1. Experimental subjects

Evaluation:
Symstra & Rostra

		Symstra				Rostra			
class	N	time	states	tests	%cov	time	states	tests	%cov
UBStack	5	0.95	22	43(5)	92.3	4.98	656	1950(6)	92.3
	6	4.38	30	67(6)	100.0	31.83	3235	13734(7)	100.0
	7	7.20	41	91(6)	100.0	*269.68	*10735	*54176(7)	*100.0
	8	10.64	55	124(6)	100.0	-	-	-	-
IntStack	5	0.23	12	18(3)	55.6	12.76	4836	5766(4)	55.6
	6	0.42	16	24(4)	66.7	-	-	-	-
	7	0.50	20	32(5)	88.9	*689.02	*30080	*52480(5)	*66.7
	8	0.62	24	40(6)	100.0	-	-	-	-
BinSearchTree	5	7.06	65	350(15)	97.1	4.80	188	1460(16)	97.1
	6	28.53	197	1274(16)	100.0	23.05	731	7188(17)	100.0
	7	136.82	626	4706(16)	100.0	-	-	-	-
	8	*317.76	*1458	*8696(16)	*100.0	-	-	-	-
BinomialHeap	5	1.39	6	40(13)	84.3	4.97	380	1320(12)	84.3
	6	2.55	7	66(13)	84.3	50.92	3036	12168(12)	84.3
	7	3.80	8	86(15)	90.0	-	-	-	-
	8	8.85	9	157(16)	91.4	-	-	-	-
LinkedList	5	0.56	6	25(5)	100.0	32.61	3906	8591(6)	100.0
	6	0.66	7	33(5)	100.0	*412.00	*9331	*20215(6)	*100.0
	7	0.78	8	42(5)	100.0	-	-	-	-
	8	0.95	9	52(5)	100.0	-	-	-	-
TreeMap	5	3.20	16	114(29)	76.5	3.52	72	560(31)	76.5
	6	7.78	28	260(35)	82.9	12.42	185	2076(37)	82.9
	7	19.45	59	572(37)	84.1	41.89	537	6580(39)	84.1
	8	63.21	111	1486(37)	84.1	-	-	-	-
HeapArray	5	1.36	14	36(9)	75.9	3.75	664	1296(10)	75.9
	6	2.59	20	65(11)	89.7	-	-	-	-
	7	4.78	35	109(13)	100.0	-	-	-	-
	8	11.20	54	220(13)	100.0	-	-	-	-

Table 2. Experimental results of test generation using Symstra and Rostra

- **“N”** : Sequences up to N methods
- **“*”** : test-generation process timed out
- **“–”** : memory limit exceeded

- **Rostra**: uses concrete arguments for state exploration
- **Symstra:** Uses symbolic execution for more comprehensive test generation
- **Evaluation Results:**
  - Symstra **generates tests faster** than Rostra
  - Symstra **explores fewer states, more relevant states** due to pruning
  - Symstra produces significantly **less number of tests** then Rostra
  - Symstra and Rostra **achieve the same branch coverage** when both don’t exceed the memory limit
  - Symstra **does not exceed the memory limit**

Conclusions & Future Directions

Key Contributions:

- **Efficient** generation of method sequences using symbolic execution.
- **Novel state comparison techniques** allowing effective pruning
- Implementation that **handles complex data structures**

Achievements:

- Symstra **generates tests faster** than traditional methods by avoiding redundant state explorations.
- **Higher branch coverage**, indicating more thorough testing

Future Directions:

- Extend Symstra to **handle concurrency**, enabling testing of multi-threaded applications.
- Improve the **handling of reference-type arguments** to extend Symstra's applicability.

Q&A