Unifying Execution of Imperative Generators and Declarative Specifications

Pengyu Nie[1], Marinela Parovic[1], Zhiqiang Zang[1],
Sarfraz Khurshid[1], Aleksandar Milicevic[2], Milos Gligoric[1]

[1] The University of Texas at Austin

[2] Microsoft

OOPSLA 2020

Grant No. 1652517

Relation of Imperative Code and Contracts

Implementations in imperative languages

```
class BST { Node root; }
class Node { int value; Node left, right;}

void add(int x) {
  if (root == null) { root = new Node(x); }
  Node n = root;
  if (n.value == x) { return; }
  else if (n.value < x) {
    if (n.left == null) { n.left = new Node(x); }
    else { n = n.left; }
  } else if (n.value > x) {
    if (n.right = null) { n.right = new Node(x); }
    else { n = n.right; }
  }
}
```

Contracts in declarative specification languages
e.g., invariants, pre-conditions, side-effects, post-conditions

invariants:
  acyclicity: $\forall\, n \in \mathrm{Node}, n \notin n(.\mathrm{left}|.\mathrm{right})+$
  order_left: $\forall\, n' \in n.\mathrm{left}(.\mathrm{left}|.\mathrm{right})*, n'.\mathrm{value} < \mathrm{n.value}$
  order_right: $\forall\, n' \in n.\mathrm{right}(.\mathrm{left}|.\mathrm{right})*, n'.\mathrm{value} > \mathrm{n.value}$

void add(int x) {
  requires: $x \notin \mathrm{root}(.\mathrm{left}|.\mathrm{right})*.\mathrm{value}$
  modifies: $\bigcup\{n.\mathrm{left}, n.\mathrm{right}, n.\mathrm{value} \mid n \in Node\} \cup \{\mathrm{root}\}$
  ensures: $x \in \mathrm{root}(.\mathrm{left}|.\mathrm{right})*.\mathrm{value}$
  $\land \forall v \in old(\mathrm{root}(.\mathrm{left}|.\mathrm{right})*.\mathrm{value}), v \in \mathrm{root}(.\mathrm{left}|.\mathrm{right})*.\mathrm{value}$
}

SAT-solver

- prototyping
- mocking
- solving NP-hard problems

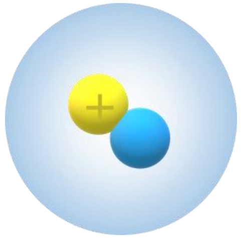
Limitations of Executable Contracts

- Traditionally written in a declarative specification language unfamiliar to most developers
  - Enable writing contracts using general-purpose languages
- Cannot benefit from partial imperative implementation ("all or nothing": an entire method is either executed declaratively or as a regular imperative method)
  - Enable fused execution of imperative code and declarative specification for a method
- Executed by translating to SAT formulas which is time-consuming and not scalable
  - Enable executing contracts using in-memory state exploration

Our Contributions

- Type-safe domain specific language for writing declarative specifications embedded in plain Java

Language ^2H



Deuterium

Framework

- Executing contracts as a combination of imperative generators and declarative specifications
  - Imperative generators offers a flexible and safe way to offload work from the solver
- Supporting two constraint solving engines
  - SAT-based solver
  - In-memory search-based solver (new)

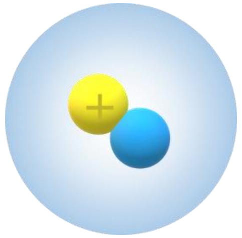
Evaluation

- Novel benchmark created using random and sequence-based test generation to mimic realistic execution scenarios
- In-memory search-based solver can be faster
- Using imperative generators speeds up execution of contracts

Our Contributions

- Type-safe domain specific language for writing declarative specifications embedded in plain Java

Language ^2H



Deuterium

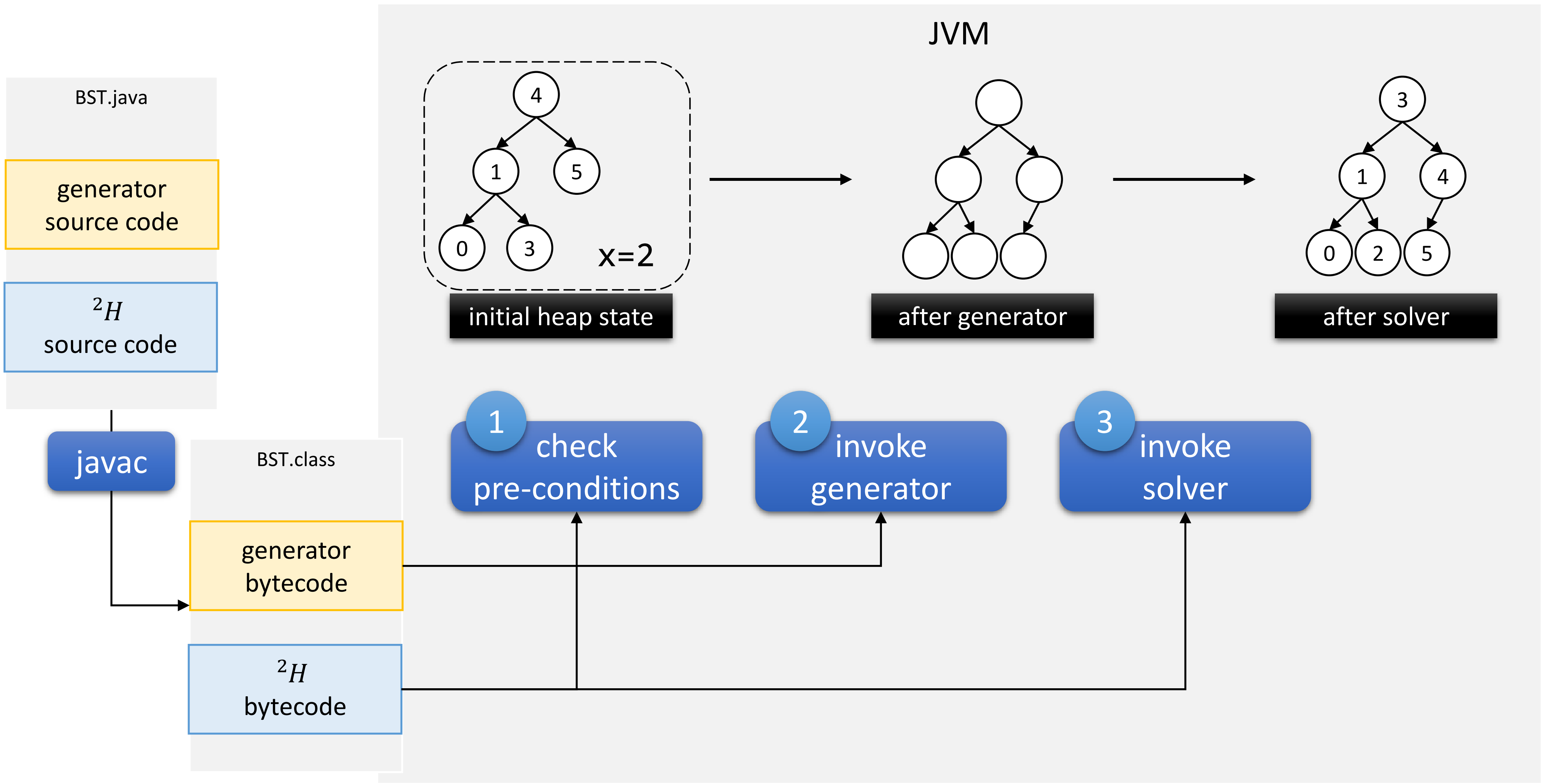
Framework

- Executing contracts as a combination of imperative generators and declarative specifications
  - Imperative generators offers a flexible and safe way to offload work from the solver
- Supporting two constraint solving engines
  - SAT-based solver
  - In-memory search-based solver (new)

Evaluation

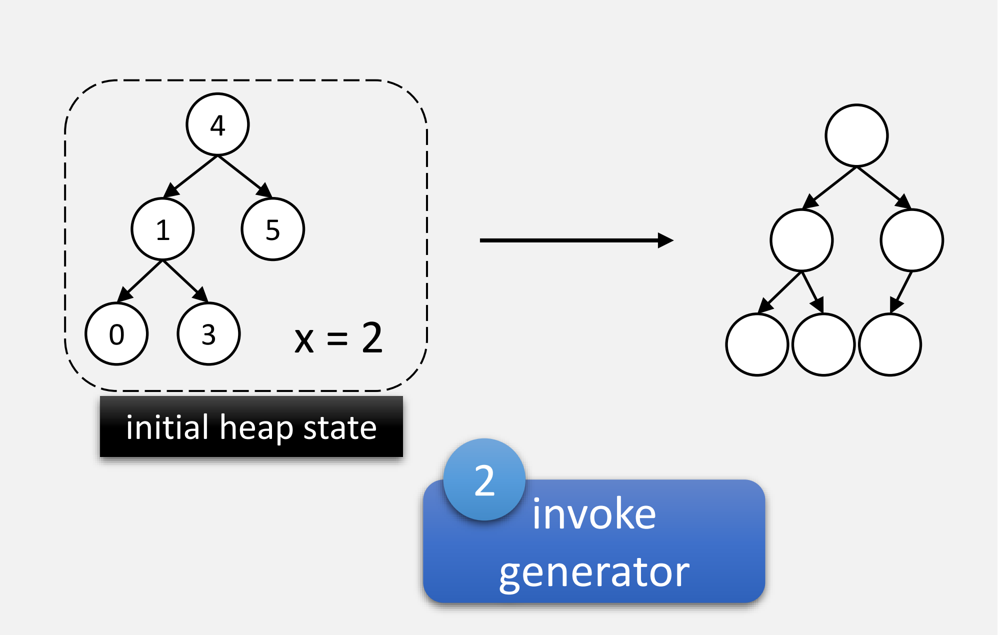
- Novel benchmark created using random and sequence-based test generation to mimic realistic execution scenarios
- In-memory search-based solver can be faster
- Using imperative generators speeds up execution of contracts

Deuterium Workflow



Imperative Generator

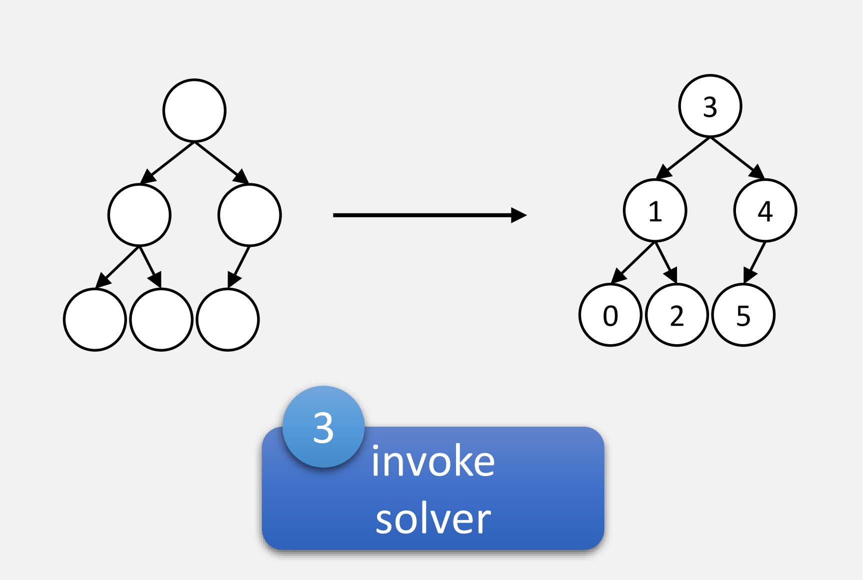
- **Usage**: Generates the backbone structure of the solution (e.g., the shape of the binary tree), offloading some work from the solver
- **Flexibility**: Users decide whether or not to use a generator and what goes into the generator
- **Safe**: If the generator contains a bug, Deuterium can detect it (because post-conditions and invariants must hold at the end of the method)




```
Node genTree(Node[] arr, int low, int high) {
  if (high <= low) return null;
  Node cur = arr[low];
  int mid = (high+low+1)/2;
  cur.left = genTree(arr, low+1, mid);
  cur.right = genTree(arr, mid, high);
  return cur; }
```

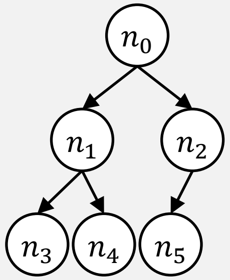
Constraint Solving Engines

- SAT-based solver
- In-memory search-based solver



SAT-Based Solver

Implemented using Squander*, with some optimizations



heap

root = $\{n_0\}$
Node_left = $\{(n_0 \to n_1), (n_1 \to n_3), (n_2 \to n_5)\}$
Node_right = $\{(n_0 \to n_2), (n_1 \to n_4)\}$
Node_old_value =
$\{(n_0 \to 4), (n_1 \to 1), (n_2 \to 5), (n_3 \to 0), (n_4 \to 3)\}$
nodes = $\{n_0, n_1, n_2, n_3, n_4, n_5\}$
old_nodes = $\{n_0, n_1, n_2, n_3, n_4\}$
x = $\{2\}$

^2H
bytecode

specifications

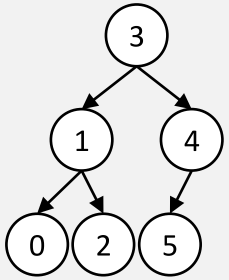
// Side-effects
Node_value = ?
// Post-conditions
nodes.Node_value = old_nodes.Node_old_value + x
// Invariants
∀n in nodes, n ⊄ n.^(Node_left + Node_right)
...

relational formula

Kodkod ⇆ SAT-solver

Node_value =
$\{(n_0 \to 3), (n_1 \to 1), (n_2 \to 4), (n_3 \to 0), (n_4 \to 2), (n_5 \to 5)\}$

relational model

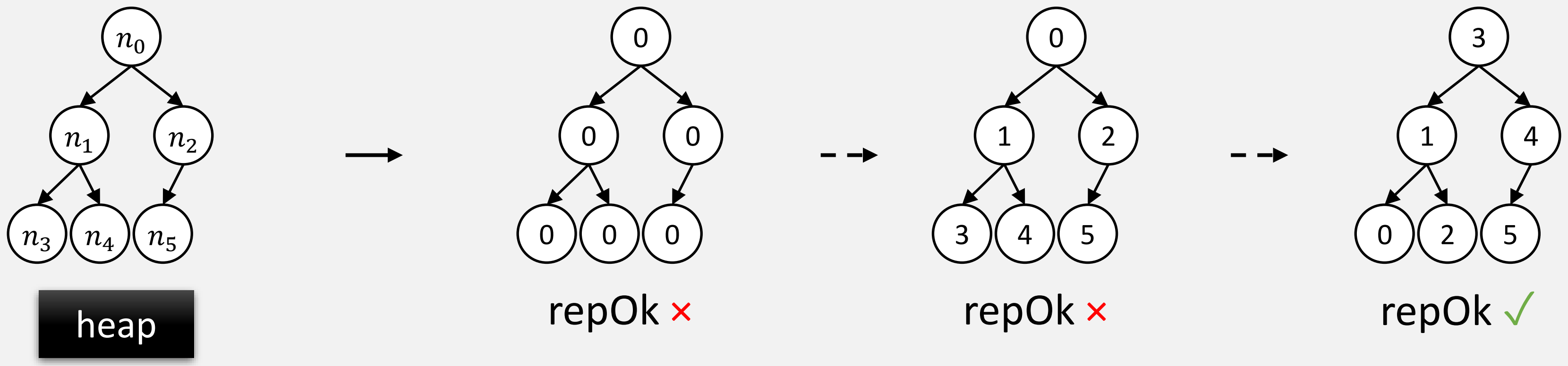


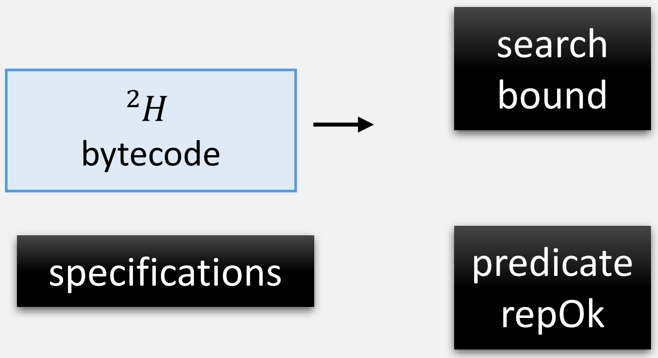
heap

* A. Milicevic, D. Rayside, K. Yessenov, and D. Jackson. Unifying Execution of Imperative and Declarative Code. In ICSE'11

In-Memory Search-Based Solver (1/2)

Exhaustively explores the search space but does so efficiently by monitoring the predicate's executions and pruning large portions of the search space*





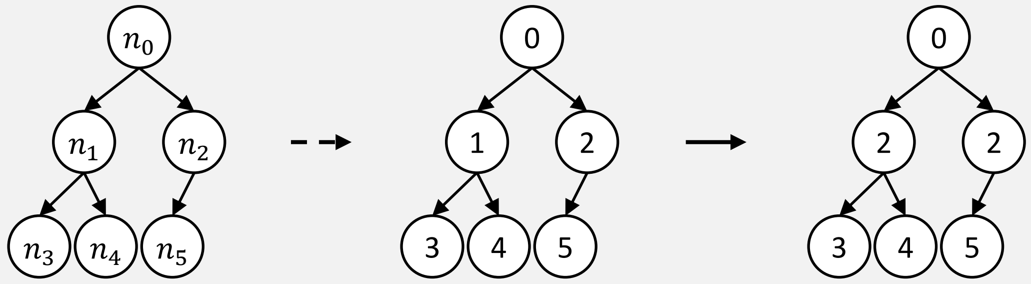
can modify: n_0.value, n_1.value, n_2.value, n_3.value, n_4.value, n_5.value
to: 0, 1, 2, 3, 4, 5

```
boolean repOk() {
  boolean invariants = nodes.all(n->n.checkInvariants());
  boolean postconds = nodes.join(n->n.value)
    .equals(old_nodes.join(n->n.old_value).union(x)));
  return postconds && invariants;
}
```

* C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated Testing Based on Java Predicates. In ISSTA'02.

In-Memory Search-Based Solver (2/2)

Exhaustively explores the search space but does so efficiently by monitoring the predicate's executions and pruning large portions of the search space*



heap

n_0.value	n_1.value	n_2.value	n_3.value	n_4.value	n_5.value
0	~~1~~ 2	2	3	4	5

predicate repOk

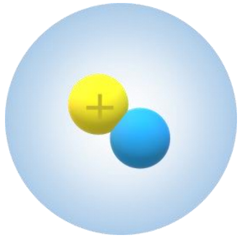
```
boolean repOk() {
  boolean invariants = nodes.all(n->n.checkInvariants());
  boolean postconds = nodes.join(n->n.value)
    .equals(old_nodes.join(n->n.old_value).union(x)));
  return postconds && invariants;
}
```

* C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated Testing Based on Java Predicates. In ISSTA'02.

Our Contributions

- Type-safe domain specific language for writing declarative specifications embedded in plain Java

Language ^2H



Deuterium

Framework

- Executing contracts as a combination of imperative generators and declarative specifications
  - Imperative generators offers a flexible and safe way to offload work from the solver
- Supporting two constraint solving engines
  - SAT-based solver
  - In-memory search-based solver (new)

Evaluation

- Novel benchmark created using random and sequence-based test generation to mimic realistic execution scenarios
- In-memory search-based solver can be faster
- Using imperative generators speeds up execution of contracts

^2H Language

```
// class Node
Rel<Node> children = D.specField(union(left, right));
D.invariant(
  closure(this, n->n.children).notContains(this),
  rclosure(left, n->n.children)
    .filter(n -> n != null).all(n -> n.value < value),
  rclosure(right, n->n.children)
    .filter(n -> n != null).all(n -> n.value > value));
// class BST
Rel<Node> nodes =
  D.specField(rclosure(root, n->n.children).subtract(null));

void add(int x) {
  if (D.specCase(nodes.join(n->n.value).notContains(x))) {
    root = genTree(nodes.union(new Node()).toArray(),
                   0, nodes.size()+1);
    D.modifies(Node.class, “value”);
    D.ensures(nodes.join(n->n.value)
      .equals(old(nodes.join(n->n.value)).union(x))); }
  D.exe(this, x); }

Node genTree(Node[] arr, int low, int high) {
  if (high <= low) return null;
  Node cur = arr[low];
  int mid = (high+low+1)/2;
  cur.left = genTree(arr, low+1, mid);
  cur.right = genTree(arr, mid, high);
  return cur; }
```

Declarative specifications in ^2H

- First order relational logic with transitive closures
- Embedded in plain Java
- Type-safe: leveraging generics

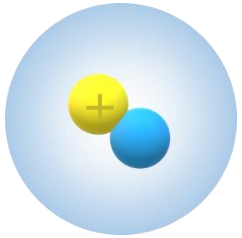
Imperative generator in Java

- Generate the backbone structure of the solution

Our Contributions

- Type-safe domain specific language for writing declarative specifications embedded in plain Java

Language ^{2}H



Deuterium

Framework

- Executing contracts as a combination of imperative generators and declarative specifications
  - Imperative generators offers a flexible and safe way to offload work from the solver
- Supporting two constraint solving engines
  - SAT-based solver
  - In-memory search-based solver (new)

Evaluation

- Novel benchmark created using random and sequence-based test generation to mimic realistic execution scenarios
- In-memory search-based solver can be faster
- Using imperative generators speeds up execution of contracts

Evaluation: Design and Subjects

- Prior work measures the time of executing a contract once
- Our workloads are intended to resemble realistic program traces: executing sequences of method calls
- Subjects: 12 data structures
  - BST, BinomialHeap, FibonacciHeap, TreeMap from JPF*
  - LinkedList, TreeMap, TreeSet from JCL (Java Class Library)
  - AvlTree, NodeCachingLinkedList, LinkedList, SinglyLinkedList, TreeSet from TACO**
- For each data structure, we write contracts for 3 methods (add, remove, find) in two ways: fusion of generator + ${}^{2}H$; pure ${}^{2}H$

* W. Visser, C. S. Păsăreanu, and R. Pelánek. Test Input Generation for Java Containers using State Matching. In ISSTA'06.

** J. P. Galeotti, N. Rosner, C. G. López Pombo, and M. F. Frias. Analysis of Invariants for Efficient Bounded Verification. In ISSTA'10.

Evaluation: Workloads

- Randomly generated workloads
  - Use Randoop to generate X tests, X = {100, 200, 500, 1000}
  - 3 methods with contracts + constructor + toString

```
public void exampleRandomWorkload1() {
  BST bst0 = new BST();
  boolean boolean2 = bst.remove(0);
  boolean boolean4 = bst.find(100);
  String str5 = bst.toString();
  assertTrue(boolean2 == false);
  assertTrue(boolean4 == false);
  assertTrue(str5.equals(“”));
}
```

- Systematically generated workloads
  - Generate all possible method sequences that lead to unique heap state up to X method calls, X = {1, 2, ..., 50}
  - 2 methods with contracts (add, remove)

```
public void exampleSystematicWorkload1() {
  BST bst = new BST();
  bst.add(0);
  bst.remove(0); }
public void exampleSystematicWorkload2() {
  BST bst = new BST();
  bst.add(0);
  bst.add(1); }
public void exampleSystematicWorkload3() {
  BST bst = new BST();
  bst.add(0);
  bst.remove(1); }
```

Evaluation: Framework Configurations

Configuration	Contract Style	Solver
Gen+SAT+CP+PI	Fusion of Generator + ${}^{2}H$	SAT-based with both CP and PI
Gen+Search+FGM		Search-based with FGM
NoGen+SAT (Squander)	Pure ${}^{2}H$	SAT-based
NoGen+SAT+CP		SAT-based with CP
NoGen+SAT+PI		SAT-based with PI
NoGen+SAT+CP+PI		SAT-based with both CP and PI
NoGen+Search		Search-based
NoGen+Search+FGM		Search-based with FGM

Optimizations to solvers (see paper):

- CP: cache parsing
- PI: performance improvement
- FGM: fine grained modifies

Evaluation: Research Questions and Setup

- Research questions
  - RQ1: Compare the execution speed of in-memory search-based solver vs. SAT-based solver?
  - RQ2: Compare the execution speed of fusion of generator + ${}^{2}H$ vs. pure ${}^{2}H$?
  - RQ3: What are the benefits obtained by Deuterium's optimizations to SAT-based solver?

see paper →

  - RQ4: How succinct are the contracts written in Deuterium?
- Setup
  - Repeat all experiments three times and report average values
  - Each run has a timeout of 30 minutes

RQ1: Search-Based vs. SAT-Based Solver

Randomly generated workloads, average of 12 data structures

#Tests	Max Heap Size	Gen+SAT+CP+PI [ms]	Gen+Search+FGM [ms]	NoGen+SAT+CP+PI [ms]	NoGen+Search+FGM [ms]
100	2.8	8,869	595	9,120	646
200	3.9	20,020	662	20,198	21,127
500	5.0	57,268	916	58,295	>30min
1,000	5.2	126,555	1,672	133,753	>30min

- Search-based solver can be faster than SAT-based solver
- However, SAT-based solver scales better when the search space is large

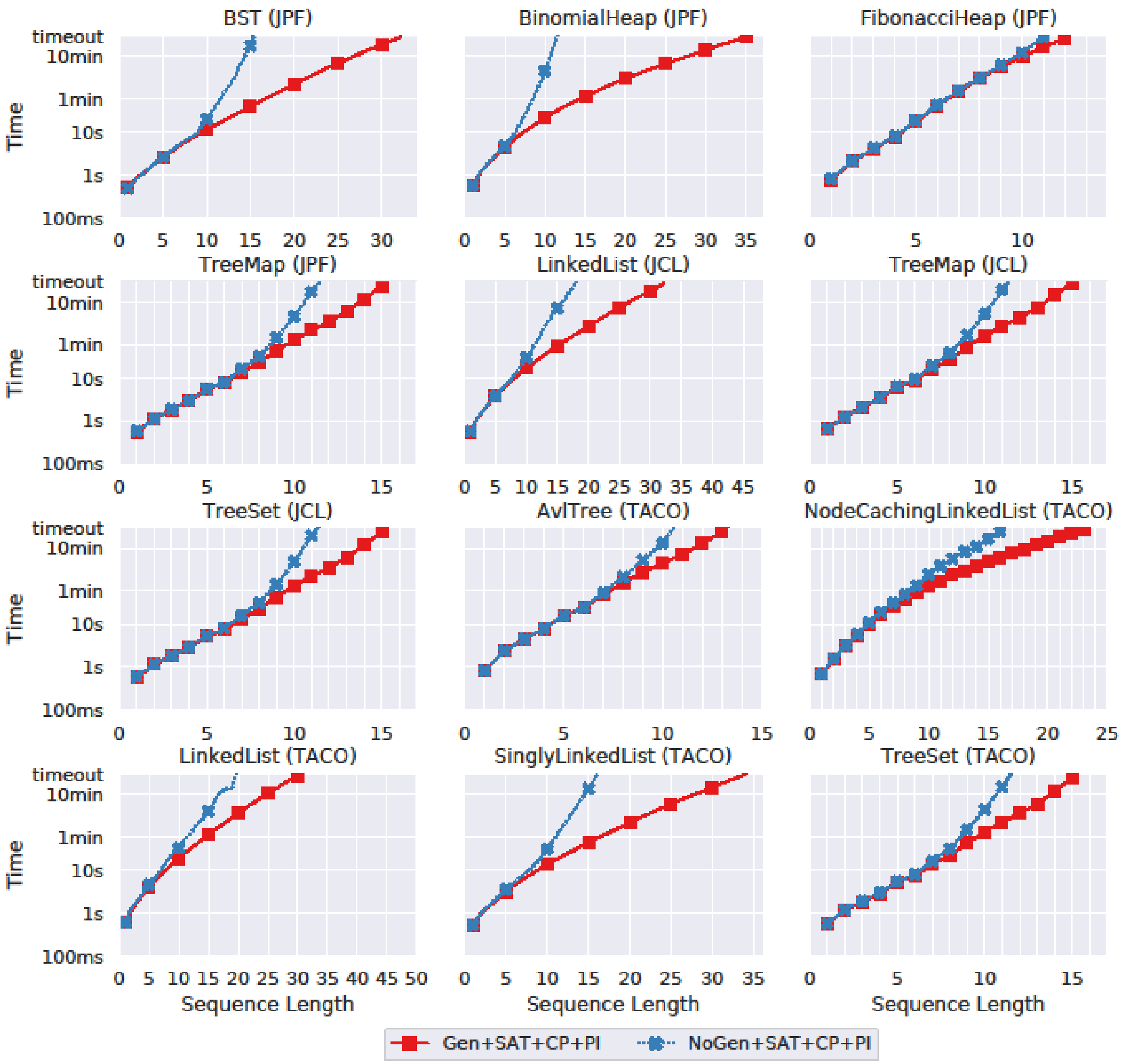
RQ2: Fusion of Generator + ^{2}H vs. Pure ^{2}H

Randomly generated workloads, average of 12 data structures

#Tests	Max Heap Size	Gen+SAT+CP+PI [ms]	Gen+Search+FGM [ms]	NoGen+SAT+CP+PI [ms]	NoGen+Search+FGM [ms]
100	2.8	8,869	595	9,120	646
200	3.9	20,020	662	20,198	21,127
500	5.0	57,268	916	58,295	>30min
1,000	5.2	126,555	1,672	133,753	>30min

- Using generator can substantially improve the performance of executing contracts for both solvers by reducing the search space

RQ2: Fusion of Generator + ^2H vs. Pure ^2H



Systematically generated workloads
x-axis: Sequence Length = Max Heap Size

- Gen+SAT+CP+PI scales for method sequences that are 5-20 method calls longer than NoGen+SAT+CP+PI

More Content in the Paper

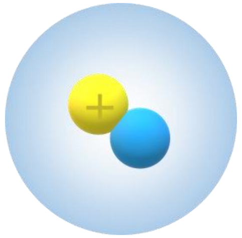
- Detailed explanation of ${}^{2}H$ language
- Evaluation results for RQ1 & RQ2 per data structure
- RQ3: benefits obtained by Deuterium's optimizations to SAT-based solver
- RQ4: succinctness of the contracts written in Deuterium
- Discussion of Deuterium's limitations

Conclusions

https://github.com/EngineeringSoftware/deuterium

Pengyu Nie <pynie@utexas.edu>

Deuterium



Language ^{2}H

- Type-safe domain specific language for writing declarative specifications embedded in plain Java

Framework

- Executing contracts as a combination of imperative generators and declarative specifications
  - Imperative generators offers a flexible and safe way to offload work from the solver
- Supporting two constraint solving engines
  - SAT-based solver
  - In-memory search-based solver (new)

Evaluation

- Novel benchmark created using random and sequence-based test generation to mimic realistic execution scenarios
- In-memory search-based solver can be faster
- Using imperative generators speeds up execution of contracts