Unifying Execution of Imperative Generators and Declarative Specifications

PENGYU NIE, The University of Texas at Austin, USA
MARINELA PAROVIC, The University of Texas at Austin, USA
ZHIQIANG ZANG, The University of Texas at Austin, USA
SARFRAZ KHURSHID, The University of Texas at Austin, USA
ALEKSANDAR MILICEVIC, Microsoft, USA
MILOS GLIGORIC, The University of Texas at Austin, USA

We present Deuterium—a framework for implementing Java methods as executable contracts. Deuterium introduces a novel, type-safe way to write method contracts entirely in Java, as a combination of imperative generators and declarative specifications (written in a first-order relational logic with transitive closure). Existing approaches are typically based on encoding both the specification and the program heap into a constraint language, and then using an off-the-shelf constraint solver—without any additional guidance—to search for a new program heap that satisfies the specification. Deuterium takes advantage of user-provided generators to prune the search space and reduce incurred overhead of constraint solving. Deuterium supports two ways of solving declarative constraints: SAT-based and search-based with in-memory state exploration. We evaluate our approach on a suite of data structures, established as a standard benchmark by prior work. Furthermore, we use random and sequence-based test generation to create a new benchmark designed to mimic realistic execution scenarios. Our results show that generators improve the performance of executable contracts and that in-memory state exploration is the algorithm of choice when heap sizes are small.

CCS Concepts: • **Software and its engineering → Specification languages**; *Imperative languages*; *Domain specific languages*; *Development frameworks and environments*; • **Theory of computation → Program specifications**.

Additional Key Words and Phrases: Imperative generators, declarative specifications, Deuterium

ACM Reference Format:
Pengyu Nie, Marinela Parovic, Zhiqiang Zang, Sarfraz Khurshid, Aleksandar Milicevic, and Milos Gligoric. 2020. Unifying Execution of Imperative Generators and Declarative Specifications. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 217 (November 2020), 26 pages. https://doi.org/10.1145/3428285

1 INTRODUCTION

A *declarative program specification* (specification for short) states the intent of the program, i.e., *what* the program is supposed to do. This stands in contrast to an *imperative program implementation*, whose main goal is to prescribe *how* this intent is to be achieved (typically by means of executing steps that are directly translatable to a commodity hardware instruction set).

Authors' addresses: Pengyu Nie, The University of Texas at Austin, USA, pynie@utexas.edu; Marinela Parovic, The University of Texas at Austin, USA, marinelaparovic@gmail.com; Zhiqiang Zang, The University of Texas at Austin, USA, zhiqiang.zang@utexas.edu; Sarfraz Khurshid, The University of Texas at Austin, USA, khurshid@utexas.edu; Aleksandar Milicevic, Microsoft, USA, almili@microsoft.com; Milos Gligoric, The University of Texas at Austin, USA, gligoric@utexas.edu.

2475-1421/2020/11-ART217
https://doi.org/10.1145/3428285

When a specification is written in a suitably expressive logic, it tends to be significantly more succinct than an equivalent imperative implementation. A common usage of specifications is for writing *method contracts* (contracts for short) [Barnett et al. 2011; Burdy et al. 2005; Chalin et al. 2005; Fähndrich et al. 2012; Kuncak et al. 2013; Logozzo 2013; Samimi et al. 2010], where a method is annotated with pre- and post-conditions expressed in a declarative logic language. Prior work mostly focused on using the contracts for runtime checking of the imperative implementations [Barnett et al. 2011; Burdy et al. 2005; Chalin et al. 2005; Fähndrich et al. 2012; Logozzo 2013].

The use of specifications in contracts gave rise to the idea of programs being written as *executable contracts*, with the promise of such programs being more readable, easier to maintain, and less error-prone than their imperative counterparts [Fuchs 1992; Hoare 1987; Kuncak et al. 2013; Polikarpova et al. 2013; Rayside et al. 2009]. Prior work explored ways of replacing imperative method implementations with executable contracts [Kuncak et al. 2013; Milicevic et al. 2011; Samimi et al. 2010], as well as adding imperative constructs around a modeling/specification language [Milicevic et al. 2014; Near and Jackson 2010]. When a contract is encountered during program execution, a suitable search algorithm (e.g., a constraint solver) is used to find a new program heap that satisfies the contract. In many cases, unfortunately, this step (translation + search) can be costly.

We propose a new approach—Deuterium—for writing contracts as a fusion of declarative specifications and imperative generators. Contracts written in Deuterium are fully executable without any additional input from the user.

The key idea behind Deuterium is that a user can provide an *imperative generator* (i.e., an arbitrary imperative code for creating and initializing objects on the heap) and use it to prune the search space and thus reduce performance overhead. Because an imperative generator is now a part of the contract, the declarative part need not be a complete functional specification; instead, it may be only a partial specification, as long as the generator and the specification together ensure the intended behavior. Furthermore, Deuterium is the first technique of its kind to demonstrate how an *in-memory state exploration* algorithm [Rosner et al. 2014] can be employed as a back-end solver; Deuterium also supports traditional SAT-based solvers.

Deuterium introduces a new specification language—^{2}H—which aims to make contracts both expressive and easy to write. (The chemical symbol for deuterium is ^{2}H.) Unlike the majority of existing specification languages (e.g., JML [Chalin et al. 2005], Spec# [Barnett et al. 2005]), ^{2}H is more expressive as it is based on a first-order logic over relations with transitive closure. Other languages that are as expressive require the specifications to be written as plain strings (e.g., JFSL [Yessenov 2009b]) or a separate domain specific language (e.g., Alloy-like language in case of PBnJ [Samimi et al. 2010]) that can be very different from the host language (e.g., Java); they also require extending the compiler (e.g., PBnJ [Samimi et al. 2010]), which might introduce additional challenges during the integration with existing development workflows. ^{2}H, in contrast, is fully embedded in Java and requires *no extension to the language syntax or compiler*, thus it benefits from many existing Java tools, from IDEs for code completion and refactoring, all the way to compilers for *static type-checking of contracts*. Other examples where a specification language is deeply embedded an imperative host language include Rosette [Torlak and Bodik 2013] for Racket, and αRby [Milicevic et al. 2014] for Ruby.

Prior work has identified *areas of applicability* for executable contracts. Concretely, tasks that are not performance critical, like prototyping, differential testing, test input generation, and mocking are all good candidates [Rayside et al. 2009; Samimi et al. 2013]; additionally, NP-hard problems where no well-known, super optimized textbook solution is readily available can sometimes be solved more efficiently with executable contracts compared to writing an ad-hoc algorithm by hand [Milicevic et al. 2011]. This paper assumes this argument and focuses entirely on improving

the state-of-the-art by (1) reducing the barrier to entry for programmers who are not necessarily proficient in formal methods by proposing an embedded, type-safe, specification language, and (2) improving scalability by using generators.

In summary, our main contributions include:

- ⋆ **Language and tool.** We introduce a type-safe language (^{2}H) and a tool (Deuterium) for writing contracts as a combination of imperative generators and specifications in a first-order relational logic with transitive closure, all embedded in plain Java. Our evaluation shows that using generators improves both performance and scalability of executable contracts.
- ⋆ **New approach to constraint solving.** Deuterium supports two constraint solving engine types: (1) SAT-based, and (2) in-memory search-based. Deuterium is the first framework to apply a search-based approach with in-memory state exploration to the problem of executing specifications. We show that the two solvers have complementary strengths.
- ⋆ **Extended benchmark suite.** The *standard benchmark* suite—established by prior work—consists of executing specifications of well-known algorithms against increasingly growing data structures and measuring how well the approach scales. We extend this suite by adding synthetic programs designed specifically to mimic realistic execution scenarios (and thus address the most common criticism of the standard benchmark). To that end, we used random [Pacheco et al. 2007] and sequence-based test generation [Visser et al. 2006] to create a *new benchmark* where methods implemented as executable contracts are executed multiple times, in a random order, as part of a larger program. We believe this new benchmark provides useful new insights and thus could prove to be a valuable tool for evaluating future approaches in this domain.

The artifacts for this paper are available at https://github.com/EngineeringSoftware/deuterium.

2 EXAMPLES

We illustrate the method contracts in Deuterium using three examples, such that each example illustrates a different aspect of the framework: (1) the binary search tree data structure, written as a combination of imperative generators in Java and declarative specifications in ^{2}H; (2) the same binary search tree data structure, but written entirely with declarative specifications in ^{2}H; and (3) an N-Queens problem solver.

2.1 Binary Search Tree: Imperative Generators + Declarative Specifications

Binary search tree (BST) is a classic data structure for storing a set of values in a sorted order. A Java representation is shown in Figure 1a: a tree has a `root` node, and each node stores a `value` and pointers to `left` and `right` subtrees.

Figure 1b shows a *declarative* specification of the BST *invariants* (lines 4–14) and *specification fields* (lines 3 and 17). Specification fields (also known as "ghost state") are defined only to simplify complex specifications, and are not used/accessed in normal execution (but only as ghost state during constraint solving). The specification field `children` (line 3) denotes both `left` and `right` children of a node. The specification field `nodes` (line 17) denotes all nodes contained in the BST. In ^{2}H, invariants are provided by virtue of calling the `D.invariant` method; Deuterium, at runtime, *asserts* that these invariants hold at the beginning of any method implemented in ^{2}H, and *ensures* that they hold at the end as well. The first invariant (line 6) specifies that the tree is acyclic: a node (`this`) must not be reachable by transitively following its `children` field. With relational algebra, the set of nodes reachable from `this` is conveniently expressed by computing the *transitive closure* of `children` and then selecting the tuples that start with `this`. The second and third invariants

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

(a) Class definitions.

```
class Node {
  // Spec field: All children
  Rel<Node> children = D.specField(union(left, right));
  { D.invariant(
     // acyclic (no node is reachable from itself)
     closure(this, n→n.children).notContains(this),
     // for each node: left subtree has lesser keys
     rclosure(this.left, n→ n.children)
        .filter(n→ n != null)
        .all(n→ n.value < value),
     // for each node: right subtree has greater keys
     rclosure(this.right, n→n.children)
        .filter(n→ n != null)
        .all(n→ n.value > value)); }}
class BST {
   // Spec field: All nodes in the tree
   Rel<Node> nodes = D.specField(rclosure(root, n→n.children).subtract(null));}
```

(b) Specifications for classes written in ^{2}H.

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

(c) Generator that creates a valid shape for BST.

```
void add(int x) {
   // SpecCase for a pre-condition
   if (D.specCase(nodes.join(n→n.value).notContains(x))) {
      root = genBalTree(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)));
   } // else nothing will be modified
   D.exe(this, x);}
```

(d) Contract for the BST.add method.

```
// class Node
@SpecField({"children: set Node | this.children = this.left + this.right"})
@Invariant({"this !in this.^children",
            "all n : {n1 : this.left.*children | n1 != null} | n.value < this.value",
            "all n : {n1 : this.right.*children | n1 != null} | n.value > this.value"})
// class BST
@SpecField({"nodes: set Node | this.nodes = this.root.*children - null"})
// method BST.add
@Specification({@Case(
   requires="x !in nodes.value",
   modifies="Node.value",
   ensures="nodes.value = @old(nodes.value) + x")})
```

(e) JFSL specifications for the BST example. In contrast to ^{2}H, JFSL specifications are written as strings in a language very different from the host language.

Fig. 1. An example contract written in Deuterium for BST.add.

(lines 8–14) specify the ordering property: values of the nodes in the left subtree are smaller than the current one, and values of the nodes in the right subtree are larger than the current one.

The `genBalTree` method in Figure 1c is an *imperative generator* used to generate a *single* BST instance (1) containing the existing nodes plus a given node, and (2) has a valid shape (the acyclicity property holds). The generator simply creates balanced tree by recursively dividing the set of nodes in two, for left and right subtrees. The idea is to use this simple generator in conjunction with the declarative specifications for `BST.add`. Consequently, the solver is now faced with a much simpler task of only assigning values to nodes (and not having to rearrange the nodes). As we will show, this improves performance.

Figure 1d shows the contract for the `add` method. A `SpecCase` provides a mechanism to split the contract into multiple cases based on different pre-conditions that may hold in the pre-invocation state. In this example, we provide a contract only when the value `x` is not already present in the tree (line 3). Otherwise, the tree does not need to be changed.

The specification in the then-branch specifies how to update the heap to include the value into the `BST`. First, `genBalTree` generates the shape for the new BST (line 4). Then, the frame condition (the `Modifies` call, line 5) specifies which fields may be modified in order to satisfy the contract. Finally, the post-condition (the `Ensures` call, on line 6), specifies the property that the solver must satisfy. Concretely, only the `Node.value` field may be modified, and the tree in the post-state must contain all the current values (`nodes.join(n→n.value)`) plus the new value `x`.

`D.exe` (line 8) executes the contract. Deuterium discovers the contract, invokes the generator, and then runs a solver to execute the specifications. A solution, if found, is guaranteed to satisfy the post-condition and all the invariants, i.e., it is correct by construction w.r.t. the provided contract. Note that the invariant on line 6 (Figure 1b) is not needed in our example because we ensure acyclicity with a generator, but we showed it for completeness.

Deuterium uses JFSL as an intermediate protocol to communicate with the SAT-based solver and introduces a novel protocol for the search-based solver. Figure 1e shows the specifications in JFSL automatically translated from the ^{2}H specifications, including the specification fields (lines 2–7), the invariants (lines 3–5), and the `SpecCase` (lines 9–12).

It is important to note that Deuterium does not prescribe any semantics for generators. Instead, Deuterium invokes the supplied generator as a regular Java method. The generator is allowed to mutate the heap in an arbitrary way. Afterwards, Deuterium attempts to satisfy the contract against the heap state left after the generator was executed. If there is a bug in the generator, we expect that constraint solving fails, because it will not be able to find a solution.

2.2 Binary Search Tree: Pure Declarative Specifications

Method contracts in Deuterium can opt to have no imperative generators, i.e., only have declarative specifications. Figure 2 shows an alternative version of contract for `BST.add` written as pure declarative specifications in ^{2}H. We use the same class definitions in Figure 1a and the same specifications for classes in Figure 1b. The `SpecCase` (line 3) checks the pre-condition such that the specification in the then-branch is only used when the value `x` is not already present in the tree. Next, the fresh object specification (line 5) specifies a new `Node` object should be created, which is referenced as `newNode`. Then, the frame condition (the `Modifies` calls, line 7–14) specifies the fields that may be modified, including `nodes`, `root`, `value` (of `newNode` and can only be modified to the new value `x`), `left`/`right` (of the nodes where the old `left`/`right` is `null`, and can be modified to either `newNode` or remain as `null`). The same post-condition (the `Ensures` call, line 16) specifies the new values should include the old values plus the value `x`.

```
void add(int x) {
  // SpecCase for a pre-condition
  if (D.specCase(!nodes.join(n→n.value).contains(x))) {
    // Fresh object
    Node newNode = D.freshObject(Node.class);
    // Modifies
    D.modifies(nodes);
    D.modifies(root);
    D.modifies(D.field(Node.class, "value"),
               (n, v)→ n == newNode, (n, v)→x);
    D.modifies(D.field(Node.class, "left"),
               (n, l)→ l == null, (n, l)→ union(newNode, null));
    D.modifies(D.field(Node.class, "right"),
               (n, r)→ r == null, (n, r)→ union(newNode, null));
    // Ensures for a post-condition
    D.ensures(nodes.join(n→n.value).equals(old(nodes.join(n→n.value).union(x))));
  } // else do nothing
  D.exe(this, x);}
```

Fig. 2. Contract for the BST.add method written in pure ^{2}H.

```
1 class Cell { int i, j; }
2 static void solveNQueens(Cell[] queens) {
3   // Requires for a pre-condition
4   D.requires(queens != null);
5   // Modifies
6   D.modifies(D.field(Cell.class, "i"));
7   D.modifies(D.field(Cell.class, "j"));
8   // Ensures for two post-conditions
9   // 1. Queens do not attack each other
10  D.ensures(range(0, queens.length).all(m→ range(m+1, queens.length).all(n→
11            queens[m].i != queens[n].i && queens[m].j != queens[n].j
12          && queens[m].i - queens[m].j != queens[n].i - queens[n].j
13          && queens[m].i + queens[m].j != queens[n].i + queens[n].j)));
14  // 2. Queens are on the chess board
15  D.ensures(range(0, queens.length).all(m→
16            queens[m].i >= 0 && queens[m].i < queens.length
17          && queens[m].j >= 0 && queens[m].j < queens.length));
18  D.exe(null, queens);}
```

Fig. 3. Contract for the N-Queens problem.

2.3 N-Queens Problem Solver

We show another example to illustrate more syntax provided by ^{2}H, in particular array operations. In this example we want to solve N-Queens problem [Wikipedia 2020], where N chess queens are placed on a N×N chessboard, such that none of the queens can attack each other. Figure 3 shows the Java definition and specifications adequate to solve the problem. A class `Cell` is defined to represent a position on the chessboard at the `i`-th row and `j`-th column. The method `solveNQueens` takes an array of `Cell` (whose length is N, i.e., `N = queens.length`) as argument, and will modify the fields `i`, `j` of the cells to get a valid solution of N-queens problem.

The pre-condition (`Requires` call, line 4) specifies that the argument `queens` should not be `null`. Deuterium will throw an exception if the requirement is not satisfied. The frame condition (`Modifies` calls, line 6–7) specifies the fields `i` and `j` of the cells can be modified.

The first post-condition (line 10–13) specifies that the queens do not attack each other. Two levels of `all` quantifications are used here, to select two (indexes of) chess queens from the array. The range of the first quantification (`m`) includes integers in the interval `[0, queens.length)`, and the range of the second quantification (`n`) includes integers in the interval `[m + 1, queens.length)`. Then it specifies that the two queens do not attack each other, i.e., not on the same row, column, or diagonal.

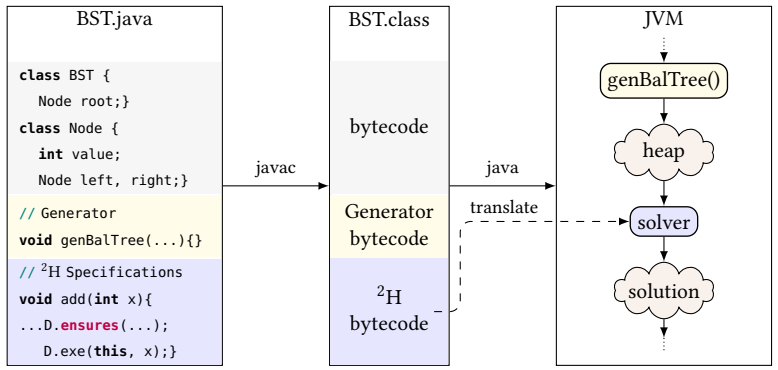

Fig. 4. Deuterium's workflow: from a contract to compiled code to invocations of the generator and a solver.

The second post-condition (line 15–17) specifies that all queens are on the chessboard of N×N, by checking if the row number `i` and the column number `j` are in the interval `[0, queens.length)`.

Finally, when `D.exe` is invoked, if there is a solution to the N-Queens problem of size N, the `queens` array will be updated to represent the solution; otherwise, Deuterium will throw an exception to report that there is no solution.

3 FRAMEWORK

This section describes Deuterium workflow (Section 3.1), ^{2}H (Section 3.2), fusion of generators and specifications (Section 3.3), integration with a SAT-based solver and our optimizations to it (Section 3.4), and an approach for utilizing search-based solver to execute specifications (Section 3.5).

3.1 Overview

Deuterium contains two major components: (1) ^{2}H for writing the declarative specifications in the contracts, and (2) a runtime system for executing the contracts. Specifications are written in first-order relational logic with transitive closure using pure Java syntax (Section 3.2) and can be combined with imperative generators (Section 3.3) for improved runtime performance and expressive power.

Figure 4 shows the workflow of Deuterium. A call to `D.exe` with the caller object and method arguments instructs Deuterium to execute the contract. Deuterium first invokes the generator and then invokes a solver with the heap and the specifications. Deuterium extracts the specifications from the compiled Java bytecode that corresponds to ^{2}H specifications and translates them into one of the formats required by the solvers, and the solvers take care of updating the program heap to reflect the solution.

Deuterium integrates with two types of solvers: (1) the SAT-based solver, which serializes the heap that becomes an input to a SAT solver and deserializes the solution, using reflection, back to JVM by changing the objects and field values on the heap, and (2) the search-based solver, which performs in-memory state exploration, so no serialization/deserialization of the heap is needed.

We implemented Deuterium as a Java library to enable the usage of the standard compiler (`javac`) to compile the entire contract to standard Java bytecode, and the usage of the default runtime (`java`) to execute the compiled bytecode. Moreover, in Deuterium, the compiler ensures that contracts are typechecked. Once `D.exe` is invoked, Deuterium takes over and performs translation of the heap to a format accepted by a solver.

Table 1. API for Relational Logic.

Type	Expression
Atomic Relations	
`Relation<T>`	`obj` `obj.type`∉{`Rel`, `Class`}, `T=obj.type`
`Relation<T>`	`obj` `Rel<T>=obj.type`
`Relation<T>`	`T.class`
`Relation<int>`	`range(int from, int to)`
`Relation<T>`	`null`
Join	
`Relation<T>`	`obj.fld` `T=fld.type`
`Relation<R>`	`join(Relation<T>, BRelation<T, R>)`
`Relation<R>`	`Rel<T>.join(BRelation<T, R>)`
Transitive / Reflexive Transitive Closure	
`BRelation<T, T>`	`closure(BRelation<T, T>)`
`Relation<T>`	`closure(Relation<T>, BRelation<T, T>)`
`BRelation<T, T>`	`rclosure(BRelation<T, T>)`
`Relation<T>`	`rclosure(Relation<T>, BRelation<T, T>)`
Set Operations	
`Relation<T>`	*op*`(Relation<T>, Relation<T>)` *op* ∈ {`union`, `subtract`, `intersect`}
Filter	
`Relation<T>`	`filter(Relation<T>, BRelation<T, boolean>)`
`Relation<T>`	`Rel<T>.filter(BRelation<T, boolean>)`
Arrays	
`Relation<int>`	`arr.length`
`Relation<T>`	`arr[idx]` `T=arr.elemType`
`BRelation<int, T>`	`elems(arr)` `T=arr.elemType`
Basic Predicates	
`boolean`	`Relation<T>` *op* `Relation<T>` *op* ∈ {`==`, `!=`, `<`, `>`, `<=`, `>=`}
`boolean`	*op*`(Relation<T>, Relation<T>)`
`boolean`	`Rel<T>.`*op*`(Relation<T>)` *op* ∈ {`in`, `notIn`, `contains`, `notContains`, `equals`}
Quantifications	
`boolean`	*q*`(Relation<T>, BRelation<T, boolean>)`
`boolean`	`Rel<T>.`*q*`(BRelation<T, boolean>)` *q* ∈ {`all`, `some`, `one`, `lone`, `none`}
Pre-invocation state	
`Relation<T>`	`old(Relation<T>)`

3.2 ^{2}H Language

^{2}H is a domain specific language for writing declarative specifications in a first-order relational logic with transitive closure. Unlike JML [Chalin et al. 2005] and JFSL [Yessenov 2009b], ^{2}H is *embedded in Java*, so new users need not learn a new syntax. ^{2}H consists of two APIs: an API for relational logic expressions, and a high-level API for specifications.

3.2.1 API for Relational Logic. This API is shown in Table 1. In each row, the left-hand side indicates the type of the right-hand side expression. We make use of the syntax for generics ⟨...⟩ to denote the relation type. We do not show integer arithmetic or boolean logical operations (e.g, &&) as they have the same semantics as in Java.

Table 2. High-Level API for Specifications.

Spec Type	API Signature
Method Specifications	
Requires	`D.requires(boolean p)`
SpecCase	`D.specCase(boolean p): boolean`
Ensures	`D.ensures(boolean p)`
Modifies	`D.modifies(Field f [, TRelation<T, R, boolean> domainFilter [, TRelation<T, R, R> rangeFilter]]) T=f.declType, R=f.type`
FreshObject	`D.freshObject(Class<T> c): T`
FreshObjects	`D.freshObjects(Class<T> c, int n): Rel<T>`
Class Specifications	
Invariant	`D.invariant(boolean... p)`
SpecField	`D.specField(Relation<T> r): Rel<T>`

In ^{2}H, all objects (except those of type `Rel`) are relations implicitly: a `Class` object represents a relation containing all instances of that class, and a non-`Class` object represents a singleton relation containing the object itself. We also introduce class `Rel`⟨T⟩ to represent explicit relations, which can be obtained via the ^{2}H API (e.g., `D.specField`), or `Rel.of`(...). `range` is a syntactic sugar that creates a relation of integers within the given bounds. `null` is a special relation that is compatible with any type.

A BRelation⟨T, R⟩ represents a binary relation of type T→R. Lambda expressions conveniently rise up to the task for expressing binary relations. For example, `t→t.fld` is a relation that maps objects of type T to their field values `fld`.

Join is a basic binary operation in relational logic. Field access (using the standard "dot" syntax in Java) on free variables is treated as a join operation. For example, `this.left` (line 8 in Figure 1b) is a join operation and is equivalent to `join(this, n→n.left)`. ^{2}H also provides the `join` API method for joins between a relation and a binary relation, where the type of the relation must match the domain of the binary relation, which is statically checked and enforced by the Java compiler.

Methods `closure` and `rclosure` compute the transitive closure and the reflexive transitive closure of a binary relation, respectively. The alternative versions of the two methods which accept two arguments—a relation and a binary relation—perform a join operation on the former and the transitive or reflexive transitive closure of the latter. These operations are particularly useful for specifying linked data structures (e.g., trees, graphs).

^{2}H supports other relational operators, including `union`, `subtraction`, `intersection`, and `filter`-ing. ^{2}H also supports getting array length and array indexing using in standard Java syntax. The method `elems` provides access to all elements of an array as a binary relation from the indexes to its elements.

Predicates in ^{2}H are expressions of boolean type, including comparisons between relations, checking if a relation is `in` (a subset of), `notIn` (not a subset of), `contains` (a superset of), `notContains` (not a superset of), or `equals` (equals to) another relation. First-order quantification is provided via the `all`, `some`, `one` (exactly one), `lone` (at most one), and `none` operators.

Finally, the `old` operation gives the pre-invocation state of a relation, i.e., the value of the relation evaluated at the beginning of the contract. This operation should only be used in post-conditions.

3.2.2 High-Level API for Specifications. This API is shown in Table 2. For each API method we show its signature. The arguments within "[]" are optional; we omit the return type if it is `void`.

The return values are used for combining the API calls with other Java code, e.g., field declaration and if-statements.

`Requires` or `SpecCase` specifies a predicate that should be checked as a pre-condition. `Requires` specifies a compulsory pre-condition which will cause execution failure if not satisfied; `SpecCase` returns a boolean value that can be used as the condition of an if-statement to represent a *specification case*: the specifications in the then-branch are used if the pre-condition is true, otherwise the specifications in the else-branch are used; we showed an example in Figure 1d.

`Ensures` specifies a predicate that must hold in the post state, i.e,. at the end of the contract.

`Modifies` specifies the side-effects the solver can make. The first argument is the field that can be modified; it can be either a field reference (e.g., `this.root`), or a Java reflections `Field` object. The second and third optional arguments allow *fine-grained* `Modifies` specifications: the second argument, `domainFilter`, specifies the instances where the field can be modified; the third argument, `rangeFilter`, specifies the possible values the field can be modified to. Both filters are ternary relations that have two arguments: the owner object (of type T) and the field value (of type R) at pre-invocation state; and one return value of type boolean or R.

`FreshObject` specifies that a new object of the argument class should be created by the solver. The return value of `FreshObject` can be used in later specifications to refer to the newly created object. `FreshObjects` should be used if multiple new objects of the same class are needed.

`Invariant`, which should be written in class initializer blocks, specifies one or more predicates that should hold for all objects of the class.

`SpecField` defines a relation that is used in other specifications to simplify formulas; the return value should be used as an initializer for a ghost field.

3.3 Fusing Generators and Specifications

The execution of purely declarative specifications might not be optimal: both a SAT-based solver and a search-based solver have the exponential worst case time complexity [Cook 1971; Levin 1973]. In contrast, many methods can be implemented imperatively with better time complexity. Using Deuterium, it is possible to fuse the two execution styles and benefit from both. Users can write an *imperative generator* to generate the backbone structure of the solution (e.g., a red black tree with a valid shape and colors without assigned values), and declarative specifications in ^{2}H to obtain the final solution (e.g., fill in the values in a red black tree). This process is illustrated in Figure 5. A solution returned by Deuterium, if found, is still guaranteed to be correct, because Deuterium asserts that all invariants and post-conditions must hold at the end of the method. If, for example, the generator contains a bug and assigns wrong colors in the generated tree, Deuterium will simply not be able to find a solution.

Note that generators in Deuterium, unlike the generators previously used in automated test generation [Daniel et al. 2007; Gligoric et al. 2010; Kuraj et al. 2015; Rosner et al. 2014], only need to generate a *single valid backbone instance* instead of all valid instances; writing a generator to obtain a single valid backbone instance is much simpler, and the execution of such generator is much faster than enumerating all instances.

Deuterium gives flexibility to users to decide whether or not to use a generator and what goes into the generator. For data structure examples like red black tree or binary search tree (Section 2.1), it is straightforward to come up with generators that generate backbone structures, and they play a key role in speeding up constraint solving (as we will show in our evaluation in Section 4). For other examples (e.g., N-Queens in Section 2.3) using generators may not provide additional benefits. Finding the right balance between generators and specifications is not straightforward and will depend on the example at hand, as well as users' background and experience.

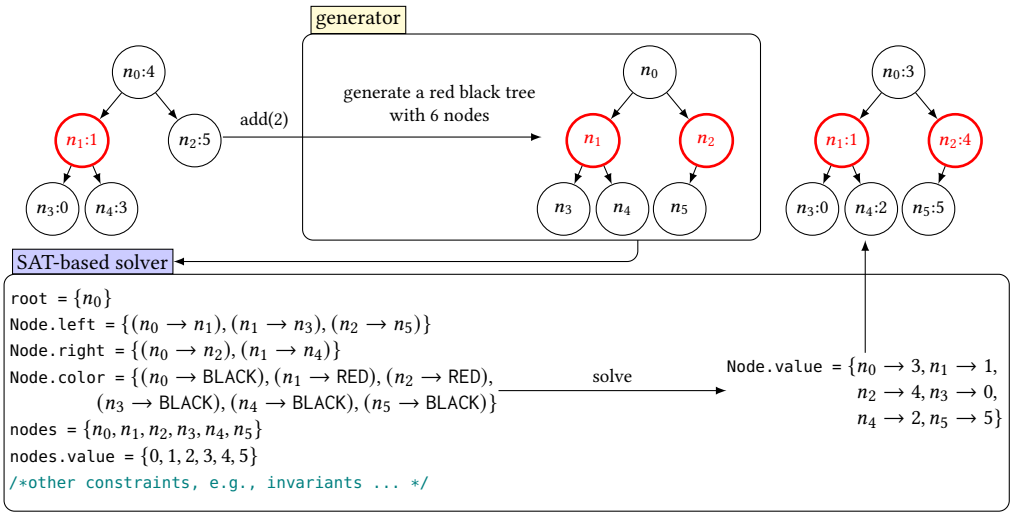

Fig. 5. Example of Deuterium adding a new value into a red black tree with a generator (top left part) and the SAT-based solver (bottom part and top right part). Deuterium uses the generator to obtain the shape of the structure and assign colors and SAT-based solver to assign values.

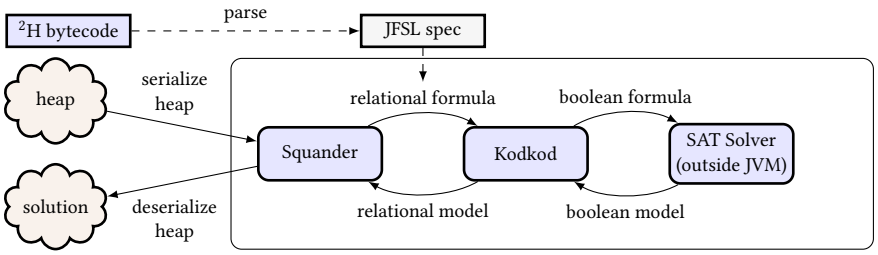

Fig. 6. Workflow of the SAT-based solver. (c.f. Fig. 4 the "solver" box.)

3.4 The SAT-Based Solver

Deuterium uses an adapted and optimized version of Squander [Milicevic et al. 2011]. As illustrated in Figure 6, Deuterium translates the specifications written in ^{2}H to JFSL, which are then solved by Squander. Note that we use the solver in an interactive mode rather than in the batch mode. Once it finds a solution, Squander automatically updates the heap to reflect the solution.

Squander [Milicevic et al. 2011] is a framework that enables solving specifications written in JFSL during the execution of Java programs. The execution of a specifications takes four steps: (1) Squander translates JFSL specifications to Kodkod [Torlak and Jackson 2007] (a constraint solver for relational logic) formulas and bounds; (2) Squander serializes the objects on the heap to Kodkod atomic relations; (3) Kodkod solves the formulas with the help of a SAT solver outside of the host JVM; and (4) if a solution is found, Squander deserializes the solution and updates the heap. Although we could have skipped the translation to JFSL and directly built Kodkod formulas, we decided to utilize Squander to simplify the implementation.

JFSL [Yessenov 2009b] is a lightweight specification language originally designed for a bounded verifier for Java called JForge [Yessenov 2009a]. JFSL specifications are written as plain-string

Table 3. Mappings from ^{2}H Operators to JFSL Specifications.

^{2}H Operator	JFSL Specification
`union`	`@+`
`subtract`	`@-`
`intersect`	`@&`
`join`	`.`
`closure`	`^`
`rclosure`	`*`
`old`	`@old`
`in`	`in`
`equals`	`=`
`filter(t, v ->r)`	`{v : t | r}`
`q(t, v ->r)`	`q v : t | r` $q \in$ {`all`, `some`, `one`, `lone`, `none`}
`range(from, to)`	`{v : int | v >= from && v < to}`
`elems(arr)`	`arr.elems`
Shared Literals and Operators	
`this`, **`null`**, `obj.fld` (field access), `==`, `!=`, `<`, `>`, `<=`, `>=`, `!`, `&&`, `||`, `arr.length` (array length query), `arr[idx]` (array indexing)	

annotations like `@Requires("...")`; syntactically, they closely resemble Alloy [Jackson 2002]. ^{2}H is as expressive as JFSL, but, in contrast, is fully embedded in Java.

3.4.1 Parsing from ^{2}H to JFSL. We built a parser from the Java bytecode of the compiled ^{2}H specifications to JFSL using the ASM library [Bruneton et al. 2002; Kuleshov 2007]. We choose to parse from the bytecode rather than the source code of ^{2}H; the latter may seem straightforward in isolation, but it takes additional steps to integrate it into compilation, store the specifications, and load them during execution. The parser reads the list of bytecode instructions in the method body or the class initializer (for class specifications). The parser identifies different types of specifications based on the APIs in Tables 1 and 2, reads the block of bytecode instructions for the API invocations, replaces the ^{2}H operators with corresponding expressions in JFSL, and outputs the result JFSL specifications. Table 3 shows the mappings used to translate ^{2}H to JFSL.

3.4.2 Cached Parsing (CP). The original solver by default collects the specifications at the beginning of each method. In cases when the same method is executed many times (e.g., in a for loop), the corresponding specifications are discovered and parsed for each method invocation. We added a caching mechanism such that the specifications are collected, parsed and cached in memory only once (at the time of the first method invocation) and are simply fetched from the cache for later invocations. This is orthogonal to caching the results of SAT solvers that are previously explored in several systems, including Green [Visser et al. 2012].

3.4.3 Performance Improvements (PI). The original Squander engine was not well suited for handling heaps with many integers. This limitation was discovered by our new benchmark based on random and sequence-based test generation. We implemented necessary optimizations, for example, speeding up the initialization of `range` integer-type relations, added them back to Squander, and observed considerable performance benefits. We believe this experience gives some more credence to the usefulness of our newly designed benchmark.

3.4.4 Example. Figure 5 illustrates the process of using Deuterium with a generator and the SAT-based solver to add a new value to a red black tree. The generator (top left part) generates a backbone of the tree with proper shape and colors; then, the SAT-based solver (bottom part)

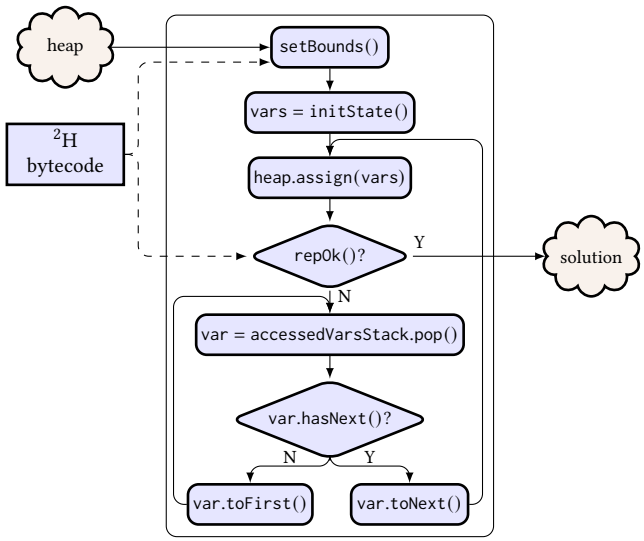

Fig. 7. Workflow of the search-based solver. (c.f. Fig. 4 the "solver" box.)

translates the current objects on the heap and the specifications written in ^{2}H to SAT formulas and solves it; finally, the heap is updated to reflect the solution (top right part).

3.5 The Search-Based Solver

Deuterium is the first approach to use an *in-memory search-based solver* to execute specifications. The main insight is that such execution can avoid the costly translation and serialization steps involved when a SAT-based solver is used. The search-based solver executes declarative specifications by performing *in-memory state exploration*, starting from the program heap at the time of the method invocation until it modifies the program heap to satisfy the invariants and post-conditions of the invoked method. To implement the in-memory state explorer we use a variant of Korat [Boyapati et al. 2002], which was originally developed for bounded-exhaustive test input generation.

Figure 7 shows the workflow of the search-based solver. Deuterium creates, from compiled ^{2}H, two methods required by the exploration algorithm: `repOk` [Liskov and Guttag 2000] for checking the invariants and post-conditions, and `setBounds` for setting the exploration boundaries from the `Modifies` specifications. The solver starts by initializing all variables to their first possible state within the boundaries (`vars = initState()`). In each iteration, the solver checks `repOk`, and if it passes, the exploration terminates with the current heap state as the solution. The solver tracks the accessed variables during the invocation and maintains a stack (`accessedVarsStack`) where variables closer to the top of the stack are accessed later. The solver then updates the state by: (a) taking the variable on top of `accessedVarsStack` (if the stack is empty, the exploration terminates with no solution) and (b) incrementing its state (i.e., changing to the next possible state within the boundary) if the variable has next state, otherwise, setting it to the first state and repeating this process. In this way, the solver prunes the search space by avoiding changing variables that are not accessed in `repOk`.

In addition to `repOk` and `setBounds`, Deuterium also creates auxiliary fields to store the pre-invocation states (i.e., old) of the fields and specification fields. Deuterium performs these translation using the ASM bytecode manipulation library [Bruneton et al. 2002; Kuleshov 2007]. We summarize the key translation steps below.

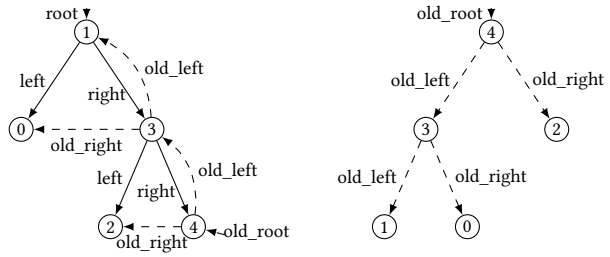

Fig. 8. Example of Deuterium creating auxiliary old fields for binary tree. On the left side is the current tree where solid arrows show current fields (root, left, right) and dashed arrows show old fields (old_root, old_left, old_right). On the right side is the old tree reconstructed from old fields. This example illustrates that copying fields by value for each object effectively does a deep copy of the entire heap.

```
1 class BST {
2  Set<BooleanSupplier> postconds;
3  boolean add() { ...
4    postconds.add(()→nodes.join(n→n.value).equals(old_nodes.join(n→n.old_value).union(x))); ...}
5  boolean repOk() {
6    return nodes.all(n→n.checkInvariants()) && postconds.allMatch(n→n.test()); }}
7 class Node {
8  boolean checkInvariants() {
9    return !( closure(this, n→n.children).notContains(this)
10      && rclosure(this.left, n→n.children).filter(n→ n != null).all(n→ n.value < value)
11      && rclosure(this.right, n→n.children).filter(n→ n != null).all(n→ n.value < value)); }}
```

Fig. 9. repOk method for executing BST.add.

3.5.1 Creating Auxiliary Fields. For each Java field (`fld`), Deuterium creates an auxiliary field (`old_fld`) to store its pre-invocation state, and updates this new field prior to executing any method contract. To save the old value, it suffices to assign `fld` to `old_fld` regardless of the field type even for reference fields, because performing this assignment for each object effectively does a deep copy; see an example in Figure 8.

For each specification field (`sfld`), Deuterium creates, from the bytecode of `SpecField` specification, an auxiliary method (`get_sfld()`) for computing the concrete values based on the program heap. During the in-memory exploration, Deuterium keeps the specification field up-to-date as the Java heap changes, i.e., when the search-based solver advances to the next heap state. Deuterium also creates an auxiliary field `old_sfld` to store its pre-invocation state.

3.5.2 Creating `repOk` *and* `setBounds`. Deuterium first creates `checkInvariants` by processing the bytecode of `Invariants` invocations; `checkInvariants` returns true *iff* the invariants of the caller object are satisfied. Deuterium maintains a set of predicates `postconds` to store the post-conditions to be checked. Deuterium modifies each `Ensures` from `D.ensures(p)` to `postconds.add(()→p)`.

Figure 9 shows the `repOk` method automatically created by Deuterium for `BST.add` from Figure 1. Deuterium creates `repOk` method as the conjunction of two predicates: (1) all invariants of the objects on the heap are satisfied using the `checkInvariants` method; and (2) all post-conditions are satisfied, which are collected in `postconds`.

Deuterium creates the `setBounds` method from the `Modifies` specifications. For each `Modifies`, Deuterium sets the bounds, i.e., the possible assignments to the fields provided in the specification, to all objects on the heap whose type matches the type of the field.

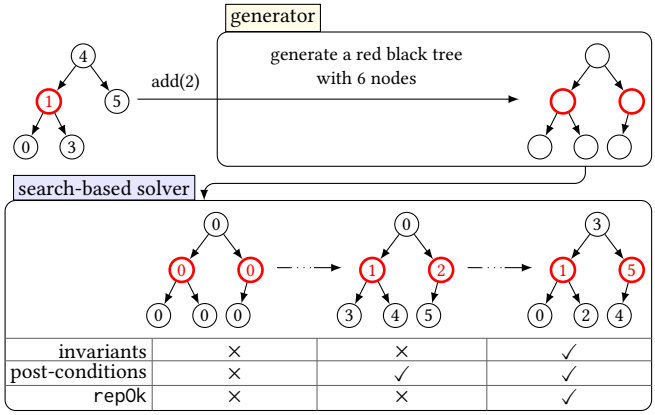

Fig. 10. Example of Deuterium adding a new value into a red black tree with a generator (top part) and the search-based solver (bottom part). Deuterium uses the generator to obtain the shape of the structure and assign colors and search-based solver to assign values.

3.5.3 Using Fine Grained Modifies (FGM). Deuterium also utilizes the fine grained modifies (i.e., `domainFilter` and `rangeFilter`) when using search-based solver. Deuterium creates a different `setBounds` method when the optional arguments to `Modifies` are provided: Deuterium sets the bound for a field of an object that do not satisfy the `domainFilter` to exactly its current value (i.e., no exploration needed); for those that satisfy the `domainFilter`, Deuterium evaluates the `rangeFilter` and uses the evaluation result as the bounds.

3.5.4 Example. Figure 10 illustrates the in-memory search exploration. The search-based solver starts from the shape and colors generated by the generator (Section 3.3), and initializes all values to `0`. Then the solver explores different heap states by modifying the values, and checking `repOk` on each heap state. The solver terminates when it finds the first heap state that satisfies `repOk` (or explores the entire space without finding a solution).

4 EVALUATION

In this section, we evaluate Deuterium and compare the performance of 8 *configurations* (Table 4). We compare two styles of writing method contracts—fusion of generator (Gen) + ^{2}H vs. pure ^{2}H—and two solvers—search-based solver (Search) vs. SAT-based solver (SAT). The configuration using pure ^{2}H and SAT-based solver with no optimization (NoGen+SAT) corresponds to the baseline in prior work—Squander [Milicevic et al. 2011].

We devise two sets of workloads that are intended to resemble realistic program traces. More precisely, we use random testing [Pacheco et al. 2007] and systematic-based test generation [Visser et al. 2006] to obtain *sequences of method calls* that differ in length; these sequences also resulted in different heap sizes. Our approach to evaluation stands in contrast to the benchmarks used in prior work where the goal was always to measure the time to execute a specification *only once* against program heaps of varying size [Milicevic et al. 2011].

We aim to answer the following research questions:

RQ1: How does in-memory exploration compare to SAT solving (in terms of execution time) on executing specifications?

RQ2: How effective are the imperative generators at improving performance and scalability of contracts?

Table 4. Configurations Used in Our Evaluation.

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

RQ3: What are the benefits obtained by Deuterium's optimizations to the SAT-based solver?

RQ4: What is the succinctness of contracts (measured in terms of the number of lines of code and number of characters) written in Deuterium compared to pure imperative code?

We next present environment setup (Section 4.1), subjects (Section 4.2), and workloads (sections 4.3 and 4.4) before answering the research questions (Section 4.5).

4.1 Environment Setup

We ran all the experiments on an Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz with 16GB of RAM, running Ubuntu 18.04 LTS. We used Oracle Java 1.8.0_191 and set a timeout to 30 minutes for each run. We repeat all experiments three times and report average values.

4.2 Subjects

We implemented several methods as contracts for 12 data structures using Deuterium. We started from data structures that are publicly available and were used in prior research [Pacheco et al. 2007; Sharma et al. 2011b; Visser et al. 2006]. Moreover, we used 3 data structures from the Java Class Library (JCL). The first column in Table 5 shows the name of each data structure and the original source: JPF [Visser et al. 2006], JCL [Oracle and/or its affiliates 2020], or TACO [Galeotti et al. 2010]. Our earlier example from Figure 1 corresponds to BST (JPF). These data structures have been establish as the standard benchmark suite in this domain.

For each data structure, we implemented three methods: `add`, `remove`, `find` (or their equivalents) with Deuterium method contracts. Imperative versions of the methods that we implemented were already available. The methods we decided to implement were frequently used as subjects in prior work on (automated) software testing, e.g., Kuraj et al. [2015]; Pacheco et al. [2007]; Sharma et al. [2011a,b]; Visser et al. [2006].

4.3 Randomly Generated Workloads

To obtain the *randomly generated workloads* we used Randoop, a tool for feedback-directed random test generation [Pacheco et al. 2007]. Randoop takes a class under test and outputs a number of JUnit tests where each test is a sequence of method calls defined by the class under test. Randoop starts by randomly choosing a method to invoke and providing default values for method arguments. In each iteration of the algorithm, Randoop chooses another method and provides as arguments either default values or the results of prior method invocations. The length of each method sequence is determined by a number of factors, including Randoop's configuration provided by the user.

In our experiments, we restrict Randoop to only use (1) constructors, (2) the `toString` method, and (3) methods that have contracts; we use default values for other configuration parameters. We obtained *four* random workloads that differ in size by running Randoop four times and providing

different limit for the number of generated tests: 100, 200, 500 and 1,000. To generate tests we used existing imperative code on which we run Randoop. We measure the *maximum heap size* of each workload, which is the maximum number of objects in the structure created at any point during the execution of the workload.

4.4 Systematically Generated Workloads

To obtain systematic workloads, we used sequence-based test generation with shape abstraction [Visser et al. 2006]. In other words, we generate tests by systematically executing *all* sequences of method calls up to a given length. We use shape abstraction to avoid sequences that lead to the same program state. The goal of our experiment was to *systematically* evaluate the impact of heap size and search space on various Deuterium configurations.

For these systematic workloads, we only use two methods (`add` and `remove` or equivalent) for each data structure to limit exponential explosion in the number of sequences; this is also consistent with prior research [Pacheco et al. 2007; Sharma et al. 2011b; Visser et al. 2006]. As a result, the heap size in each workload correlates with the length of sequences. For each data structure, we iteratively extend the max sequence length up to 50 (and 35 for FibonacciHeap (JPF) because test generation time exceeds 12 hours) and as long as there is at least one Deuterium configuration that does not exceed execution time of 30 minutes for generated sequences.

4.5 Results

Table 5 shows the results of our experiments for various configurations under randomly generated workloads. The first column shows the name of the data structure, the second column shows the size of the workload (i.e., number of generated tests), and the third column shows the maximum heap size of the workload. Columns 4–11 show the time to execute the workload with 8 configurations. We highlight the best/shortest time for each workload using bold fonts. All reported numbers for execution time are in milliseconds.

Figure 11 shows the results for the systematic workloads, one line plot for each data structure. We focus on comparing Gen+SAT+CP+PI and NoGen+SAT+CP+PI configurations as these two configurations provide the best view of the improvement over the state-of-the-art. In each plot, x-axis shows sequence length and y-axis (log scale) shows *cumulative* (i.e., time for all sequences in the workload) execution time. For example, TreeMap (JCL) has 16,524 sequences at length 18. Note that both the time needed to execute one sequence and the number of sequences grow exponentially as sequence length increases.

Figure 12 shows the results for the systematic workloads, in the same format as the aforementioned results, but shows the execution time divided by the number of sequences in the workload. The discontinued lines means the total execution time exceeds the time limit we set (30 minutes).

We can see that for NoGen+SAT+CP+PI, the execution time per sequence grows exponentially as the size of the sequence increases. In contrast, for a number of subjects, the execution time per sequence of Gen+SAT+CP+PI grows linearly. The results strengthen our findings that using generators in contracts greatly reduces execution time and improves scalability.

Answering RQ1. Our results in randomly generated workloads (Table 5) show that Gen+Search+FGM always outperforms Gen+SAT+CP+PI by several orders of magnitudes. For all subjects but FibonacciHeap, NoGen+Search+FGM outperforms NoGen+SAT+CP+PI. This indicates that doing in-memory state exploration to solve declarative specifications *can be* faster than invoking a SAT solver.

Another observation is that for larger workloads, SAT-based solver scales better with the increase of the heap size. (We made this observation in our preliminary evaluation not reported in this

Table 5. Number of Tests, Maximum Heap Size and Test Execution Time (in Milliseconds) for Randomly Generated Workloads.

Class	#Tests	Max Heap	Gen +SAT +CP+PI	Gen +Search +FGM	NoGen+SAT -	NoGen+SAT +CP	NoGen+SAT +PI	NoGen+SAT +CP+PI	NoGen+Search -	NoGen+Search +FGM
BST (JPF)	100	2	3,303	647	5,205	3,968	4,511	3,277	711	**561**
	200	4	6,095	**623**	10,037	7,623	8,484	5,973	887	630
	500	5	13,725	**702**	24,611	18,686	19,445	13,216	28,448	704
	1000	5	27,406	**961**	51,481	38,566	38,817	25,938	29,870	1,035
BinomialHeap (JPF)	100	2	6,535	**541**	9,379	7,045	8,847	6,501	545	588
	200	2	13,119	**617**	19,588	14,967	18,194	13,155	651	645
	500	4	35,946	**698**	54,778	42,500	48,291	35,485	1,651	917
	1000	4	77,952	**906**	120,466	92,847	104,192	76,128	58,214	1,893
FibonacciHeap (JPF)	100	5	28,673	**589**	34,997	30,625	33,486	30,489	1,156	836
	200	6	79,449	**727**	91,215	82,549	88,903	78,146	939,386	244,972
	500	8	257,342	**2,421**	300,317	269,770	290,658	258,851	timeout	timeout
	1000	8	584,470	**8,170**	723,838	647,229	694,799	658,475	timeout	timeout
TreeMap (JPF)	100	3	6,149	**526**	9,302	7,029	8,678	6,140	856	610
	200	3	12,370	**611**	19,938	14,525	17,660	12,253	1,150	666
	500	4	32,126	**691**	53,918	39,934	46,716	32,161	23,704	990
	1000	5	68,372	**1,022**	119,597	87,747	100,095	68,439	446,545	2,818
LinkedList (JCL)	100	3	5,063	**612**	7,138	5,899	6,511	5,212	688	703
	200	4	9,928	**715**	14,571	11,747	12,941	9,835	4,293	962
	500	5	25,273	**915**	38,206	31,178	32,589	25,384	187,966	1,295
	1000	5	53,825	**1,271**	80,366	65,513	67,475	53,240	829,906	4,270
TreeMap (JCL)	100	3	7,949	**610**	13,042	10,663	10,367	8,692	1,067	651
	200	4	16,829	**725**	29,604	24,266	22,379	17,930	8,496	821
	500	5	44,526	**881**	80,722	66,051	59,223	44,566	227,391	3,597
	1000	5	98,366	**1,335**	177,956	146,848	129,757	98,190	timeout	8,121
TreeSet (JCL)	100	2	6,093	**612**	9,262	6,569	8,590	5,972	645	632
	200	4	12,016	**649**	19,402	13,698	17,448	11,934	2,270	822
	500	5	31,725	**813**	52,170	37,427	46,694	31,167	821,340	2,283
	1000	5	66,922	**1,095**	113,551	79,671	99,072	67,536	804,446	2,677
AvlTree (TACO)	100	3	19,180	**544**	21,588	20,156	20,739	19,086	1,070	666
	200	4	43,573	**597**	49,411	45,607	46,541	43,810	5,433	851
	500	4	123,624	**719**	141,855	132,677	134,410	125,838	47,336	1,652
	1000	5	278,792	**1,029**	311,358	297,287	298,140	273,176	148,082	4,055
NodeCaching-LinkedList (TACO)	100	3	8,017	**670**	10,872	8,839	10,266	8,118	732	699
	200	4	16,707	**722**	23,616	19,479	21,866	17,633	903	967
	500	5	45,399	**895**	66,429	55,858	61,122	49,533	1,711	1,701
	1000	5	98,905	**1,213**	146,786	120,769	132,177	107,858	3,488	3,513
LinkedList (TACO)	100	3	5,518	**623**	8,050	6,467	7,501	5,646	714	642
	200	4	10,665	**688**	16,717	13,439	14,918	11,594	1,194	713
	500	5	27,707	**817**	45,854	37,427	39,789	31,776	6,586	844
	1000	5	59,804	**1,112**	98,567	81,009	85,813	67,729	16,247	1,171
SinglyLinkedList (TACO)	100	3	4,020	**514**	6,212	4,971	5,675	4,340	547	542
	200	4	7,558	**566**	12,580	9,872	10,950	8,237	607	597
	500	5	18,424	**653**	32,367	26,094	26,937	20,542	703	684
	1000	5	37,855	**852**	68,348	55,775	55,501	42,786	983	904
TreeSet (TACO)	100	2	5,931	653	8,601	6,571	8,159	5,978	665	**623**
	200	4	11,932	**705**	17,862	13,642	16,068	11,879	3,554	887
	500	5	31,408	**797**	48,009	36,762	42,575	31,021	1,246,943	1,972
	1000	5	65,997	**1,098**	102,833	78,969	91,421	65,542	1,247,772	2,586

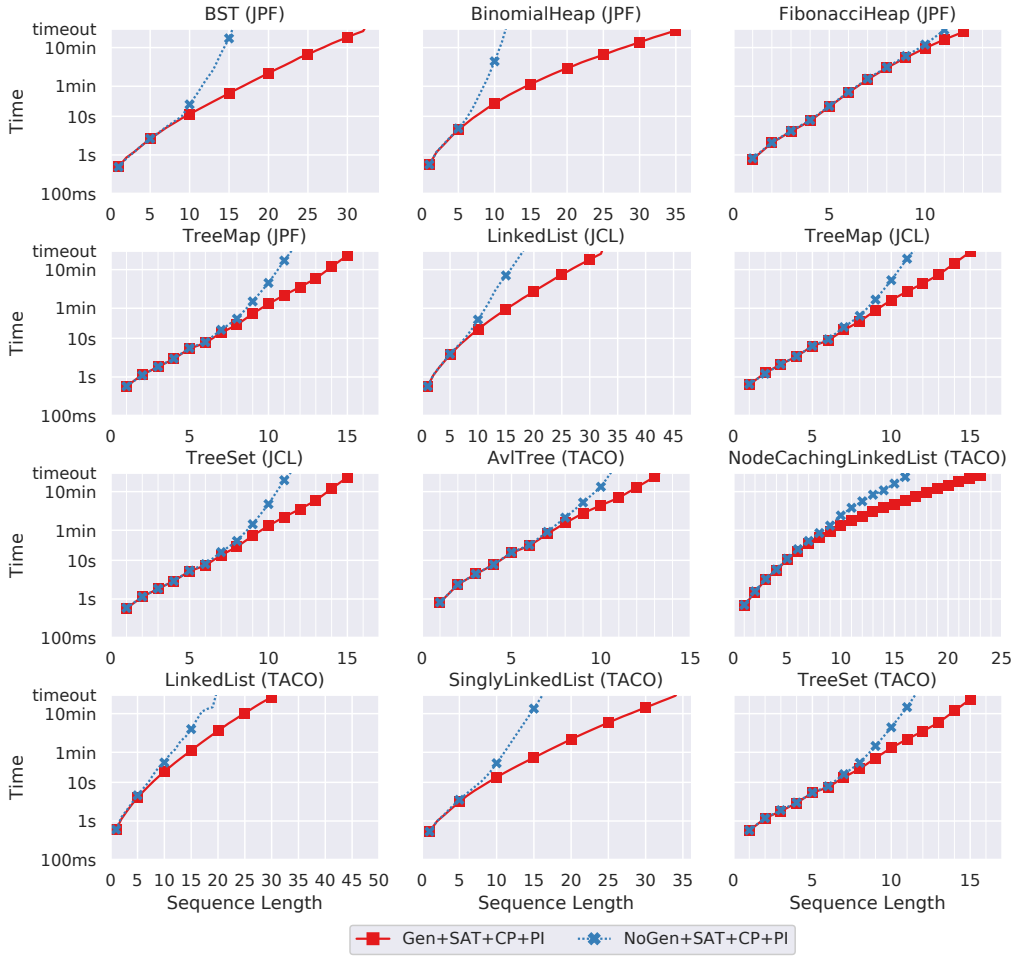

Fig. 11. Execution time in systematically generated workloads for two Deuterium configurations.

paper.) Therefore, we used a SAT-based solver to evaluate the effect of using generators (RQ2) in systematically generated workloads which have much larger heap sizes.

Answering RQ2. Results for both workloads show that using generator can substantially improve the performance of executing contracts, especially in the execution with larger heap sizes. In Table 5, Gen+Search+FGM is the best configuration for all but two small workloads on BST (JPF) and TreeSet (TACO). In Figure 11, Gen+SAT+CP+PI achieved better performance than NoGen+SAT+CP+PI on all subjects.

Regarding the scalability, based on the results for systematically generated workloads, we can see that Gen+SAT+CP+PI scales for method sequences that are 5–20 method calls longer than NoGen+SAT+CP+PI.

Answering RQ3. We can observe that both optimizations (CP, PI) for SAT-based Deuterium had positive effect on speeding up the constraint solving. Based on the numbers in Table 5, NoGen+-SAT+CP+PI saves 31.34% of execution time on average (up to 49.62%) compared to NoGen+SAT, while NoGen+SAT+CP saves 19.66% (up to 29.84%) and NoGen+SAT+PI saves 11.52% (up to 27.08%). These differences are quite significant, which clearly illustrates the benefits of our optimizations.

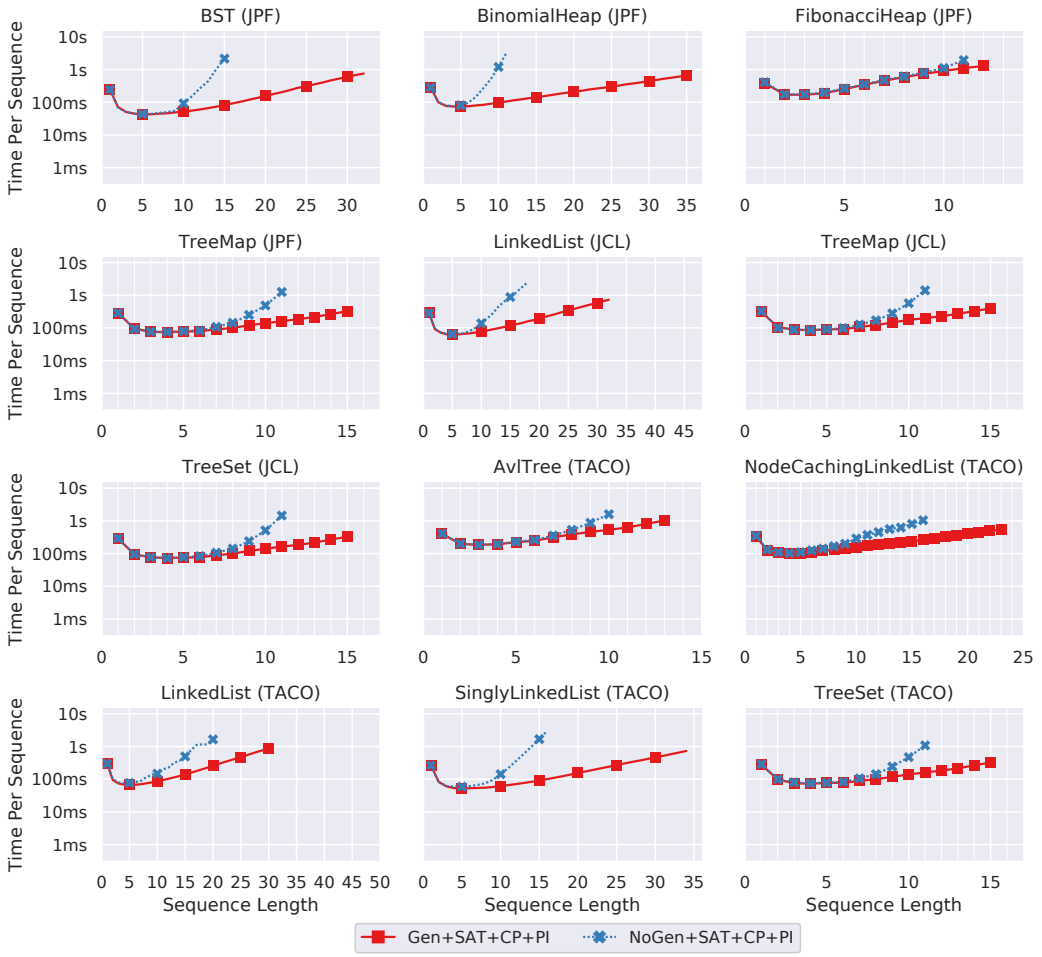

Fig. 12. Execution time per sequence in systematically generated workloads.

Answering RQ4. To answer this question, we compute lines of code (LOC) and number of characters (NOC) for the methods (or contracts) used in our study and code they depend on. We used NOC in addition to LOC because declarative specifications and imperative code have different formatting styles.

Table 6 shows our results. On average, contracts in ^{2}H has LOC of 115 and NOC of 3,164; fusion of generators and ^{2}H has LOC of 161 and NOC of 3,973; while equivalent imperative implementations has LOC of 284 and NOC of 4,455. ^{2}H shows succinctness over imperative implementations on many of the subjects (e.g., in TreeMap (JPF): LOC 134 vs. 460, NOC 3,387 vs. 6,668). The imperative implementation of these data structures can be hard and non-trivial, thus the declarative specifications in ^{2}H can be used as (executable) prototypes at an early stage of development.

5 LIMITATIONS AND FUTURE WORK

An advantage of the search-based solver is that it supports some Java features, e.g., floating point computation, that are not well supported by SAT-based solvers [Alouneh et al. 2018]. Studying

Table 6. Comparison of LOC and NOC (Number of Characters) among Pure ^{2}H, Gen+^{2}H (Fusion of Generators + ^{2}H), and Imperative Implementation.

	Class	^{2}H		Gen+^{2}H		Imp.	
		LOC	NOC	LOC	NOC	LOC	NOC
JPF	BST	83	1,920	110	2,409	157	2,161
	BinomialHeap	152	4,401	194	5,035	325	5,329
	FibonacciHeap	152	4,355	212	5,434	268	3,813
	TreeMap	134	3,387	193	4,532	460	6,668
JCL	LinkedList	102	2,730	122	2,952	304	4,572
	TreeMap	109	3,291	174	4,695	510	8,178
	TreeSet	140	3,578	198	4,725	475	7,484
TACO	AvlTree	99	2,653	139	3,415	188	3,533
	NodeCachingLinkedList	135	4,084	194	4,923	178	3,149
	LinkedList	97	2,682	130	3,103	126	2,109
	SinglyLinkedList	74	1,832	103	2,250	78	1,130
	TreeSet	105	3,059	163	4,206	349	5,345
Avg.		115	3,164	161	3,973	284	4,455

examples that require such features is an interesting direction to explore. This will lead to further fusions: a user could combine an imperative generator, a declarative specification to be executed by the search-based solver, and a declarative specification to be executed by the SAT-based solver.

Our results showed that Deuterium with the search-based solver outperforms Deuterium with the SAT-based solver if the heap sizes are small. Deuterium could *automatically choose* the most appropriate solver for each invocation of a method contract based on prior invocations with similar heap shapes and sizes.

Based on our anecdotal experience and empirical evidence (answers to RQ4 in Section 4), writing a contract as a combination of generators and specifications is usually easier and less error-prone than writing a full imperative implementation. Future work could perform a user study to collect more empirical evidence to support this finding. Specifically, we could design a controlled experiment where two groups of users are asked to complete the same programming task, where one group writes contracts in Deuterium and the other group writes imperative code (in Java); the two groups of users should have similar experience on programming in Java and be given the same tutorials about the task and Deuterium. Then, we could measure the time, LOC, and NOC to complete the task for each user, and check the correlation between these measurements for the two groups.

The user interaction with Deuterium could be improved by providing enhanced debugging experience. For example, if the user-provided generator has a bug, Deuterium currently prints an error message when constraint solving fails to find a solution. In the future, we plan to localize the bug and point to specific lines in the generator that are incorrect.

Ideally, we want Deuterium to enable users to obtain both *correct code and efficient implementation*. Currently, Deuterium ensures correctness by enabling the execution of contracts; however, the execution cost might not be acceptable for a production environment (although it should be fine for non-performance-critical tasks). To improve performance, the future work should develop an automated synthesis technique that transforms contracts written in Deuterium—including generators and specifications—to fast imperative implementations. Additionally, the synthesis technique should produce a certificate that the contract and the output imperative implementation are consistent. Rayside et al. [2012] did a preliminary exploration in this direction, as they synthesized imperative iterators for JDK Collections classes from declarative specifications.

6 RELATED WORK

We present the most closely related work. We compare our work to two groups: (1) prior work on integration of declarative and imperative paradigms, and (2) declarative languages for Java.

Integration of declarative and imperative paradigms. Lopez et al. [1993] introduced Kaleidoscope, a tool for Constraint Imperative Programming, which integrates constraints and object-oriented programming. JForge Specification Language (JFSL) [Yessenov 2009b] is a language that combines Alloy-like first-order relational logic with transitive closure and Java. It was developed for JForge [Yessenov 2009a,b], which is a bounded verifier. The design of ^{2}H is partially inspired by JFSL; ^{2}H uses pure Java syntax to achieve equivalent expressiveness level for declarative specifications. Our contributions also include combination of generators and specifications.

Squander [Milicevic et al. 2011] enabled the execution of embedded JFSL specifications in Java; the specifications are written in Java annotations. Deuterium adapted and optimized Squander as a SAT-based solver for executing specifications written in ^{2}H.

Kuncak et al. [2013] studied executing specifications using synthesis and constraint solving for Scala. Several differences between their work and our work include: (1) on the language side, Scala supports functional programming and is more extensible, thus the integration with declarative specifications is simpler; (2) they compile as many specification fragments as possible to functional programming code before invoking a constraint solver, and in contrast, we proposed using generators to limit the search-space; (3) their work is backed by a conventional SMT solver, while our work is the first to use in-memory search to execute contracts (in addition to using a SAT-based solver); (4) we introduce a novel benchmark that mimics a more practical scenario.

PBnJ [Samimi et al. 2010] provides an extension to Java to support runtime checking using declarative specifications, and it also supports replacing the execution of imperative implementation with the execution of declarative specifications when the former is found to be incorrect. It achieved so by extending the Java syntax and Java compiler; in contrast, ^{2}H requires no extension to syntax/compiler to support a fusion of declarative and imperative execution in Java. One can use Deuterium to re-implement the functionalities provided by PBnJ.

The Alloy modeling language [Jackson 2002] is a declarative language based on first-order relational logic with transitive closure. Prior work studied the integration of Alloy and Java. Al-Naffouri [2004] developed MintEra, a framework that transforms the Alloy predicates (written as comments in Java) to executable Java code using a customized Java parser. Rosner et al. [2014] developed HyTek that combines declarative predicates in Alloy and imperative predicate methods in Java to achieve more efficient test generation. TestEra [Marinov and Khurshid 2001] is a framework for automated specification-based testing with Alloy specifications using Alloy analyzer. Deuterium is the first approach for writing method contracts as a combination of generators and declarative specifications. Additionally, ^{2}H uses a pure Java syntax.

Meng et al. [2017] developed an extension for CVC4, which enables mapping of a declarative modeling language (Alloy) to SMT constraints. More recently, Abbassi et al. developed Astra [Abbassi 2018; Abbassi et al. 2019], a library that converts Alloy to SMT-LIB. Future work could enable Deuterium to utilize SMT solvers.

There is a closely related line of work on unifying imperative generators and declarative specifications for (random) test generation. UDITA [Gligoric et al. 2010] is a Java-like language with non-deterministic constructs for writing test input generators. SciFe [Kuraj et al. 2015] is a framework for combining enumerators aimed for exhaustively generating instances of complex data structures. Claessen et al. [2014] proposed an approach to write generators that conform to "almost uniform" distribution with respect to the given declarative predicates. Fetscher et al. [2015] built a solver that effectively generates random well-typed test inputs, given a specification of

a type-system expressed in a subset of first-order logic with equality and inequality constraints. Luck [Lampropoulos et al. 2017] is a domain-specific language and framework to simplify writing and maintenance of generators; the generators are conveniently expressed by decorating predicates with lightweight annotations to control the distribution of generated values and the amount of constraint solving that happens before each variable is instantiated. Prior work reduces the problem of generating random test inputs to constraint solving. But unlike Deuterium, which starts from an already existing heap, they start from an empty heap and generate *many* heaps that satisfy the specification. The goal of generators in Deuterium is to obtain only a *single* valid instance (which is an easier task).

PQL [Reichenbach et al. 2012] is a first-order query language embedded in Java for expressing parallel computations. PQL targets specific domain and does not integrate imperative generators.

Prior work also studied the integration of declarative specifications into other imperative programming languages. Ball and Rajamani [2001] developed SLIC, a specification language for checking pre- and post-conditions for C. CodeContract [Fähndrich et al. 2012, 2010; Logozzo 2013] is a plugin for .NET to support writing method specifications in place for runtime checking and visualization. Barnett et al. [2011, 2005] developed Spec#, an extension of .NET, that enables writing specifications about methods and data usages. Unlike prior work, Deuterium targets implementation via specification, although the specifications could be used for runtime checking.

Torlak and Bodik [2013] developed Rosette, a framework for designing and developing solver-aided languages, i.e., to introduce syntax for declarative specifications into an existing imperative language. The goal of Deuterium, however, is to integrate declarative pattern into Java without changing syntax.

Sketch [Solar-Lezama et al. 2006] is a system for program synthesis, i.e., finding a program that satisfies a specification for *all* possible heaps. Deuterium finds *one* post-heap that satisfies a specification for a given pre-heap. Also, Deuterium brings new language embedded in Java and combination of generators and specifications for a different task (i.e., executable method contracts).

Declarative languages for Java. JML [Burdy et al. 2005; Chalin et al. 2005] is a specification language used to formally specify the behavior of Java classes; the specifications are based on first-order logic with quantifiers. Specifications in JML are written as comments and additional tools are required to utilize them, e.g., OpenJML [Cok 2011]. Chalin et al. [2010] discussed and envisioned various applications of JML. In particular, they prototyped Java Contract for writing JML-like specifications as code; however, their API was rather simple as opposed to ^{2}H which is based on relational logic.

7 CONCLUSION

We presented Deuterium, a framework for implementing Java methods as executable contracts. We introduced a novel, type-safe way to write method contracts entirely in Java as a combination of imperative generators and declarative specifications. Deuterium also introduced an in-memory state exploration algorithm designed to take advantage of user-specified imperative generators to prune the search space and reduce incurred overhead. We evaluated this novel approach on a suite of standard data structures. Additionally, we used random and sequence-based test generation to create a new benchmark designed to mimic more realistic execution scenarios. Our results showed that in-memory state exploration is an ideal choice if heap sizes are small while the generators substantially improve performance of executable contracts in general. We believe that Deuterium enables the use of method contracts for non-performance critical tasks, like prototyping, differential testing, test input generation, and mocking.

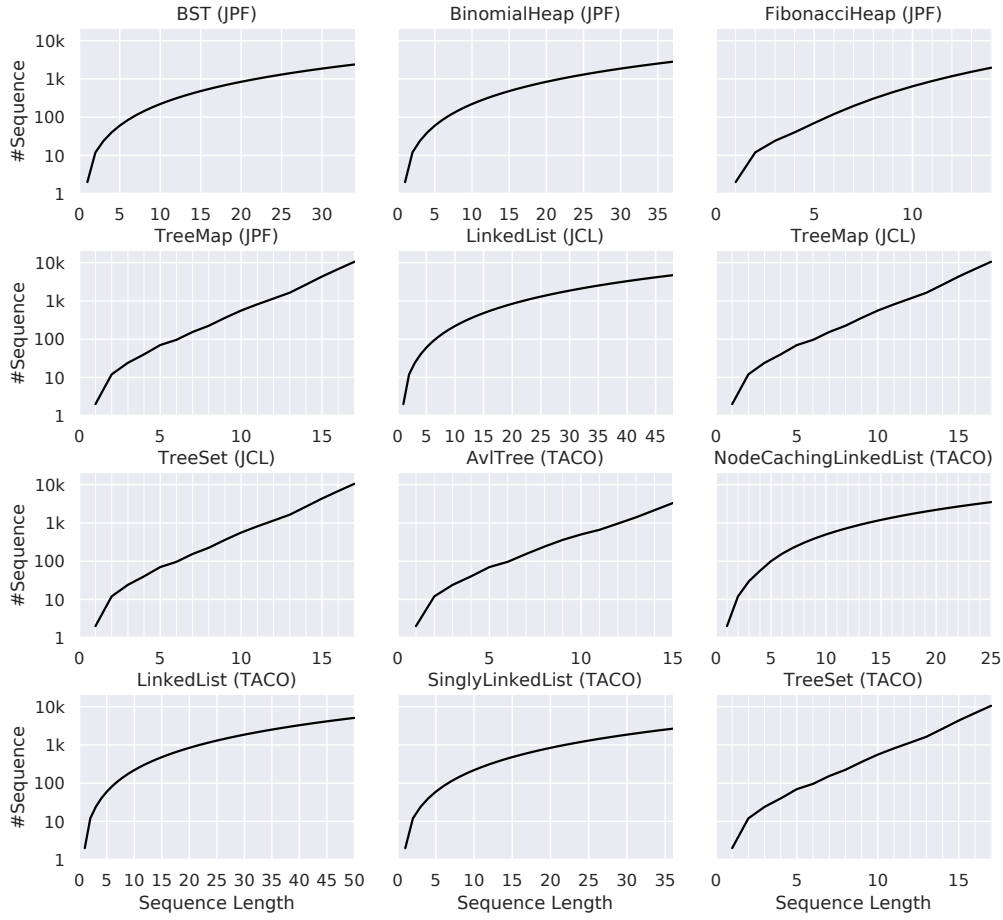

Fig. 13. Number of sequences in systematically generated workloads.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments. We also thank Ahmet Celik, Karl Palmskog, Nadia Polikarpova, and Chenguang Zhu for their feedback on this work. This work was partially supported by the US National Science Foundation under Grant No. 1652517.

A APPENDIX

Figure 13 shows the number of systematically generated sequences for the data structures at each size (up to the sizes used in our evaluation).

REFERENCES

Ali Abbassi. 2018. Astra: Evaluating Translations from Alloy to SMT-LIB. http://hdl.handle.net/10012/14286

Ali Abbassi, Nancy A. Day, and Derek Rayside. 2019. Astra Version 1.0: Evaluating Translations from Alloy to SMT-LIB. *ArXiv* abs/1906.05881 (2019).

Basel Y. Al-Naffouri. 2004. *MintEra: A Testing Environment for Java Programs.* Thesis (M. Eng.). Massachusetts Institute of Technology.

Sahel Alouneh, Sa'ed Abed, Mohammad H. Al Shayeji, and Raed Mesleh. 2018. A Comprehensive Study and Analysis on SAT-Solvers: Advances, Usages and Achievements. *Artificial Intelligence Review* (2018), 1–27.

Thomas Ball and Sriram K Rajamani. 2001. *SLIC: A Specification Language for Interface Checking (of C).* Technical Report. Technical Report MSR-TR-2001-21, Microsoft Research.

Mike Barnett, Manuel Fähndrich, K Rustan M Leino, Peter Müller, Wolfram Schulte, and Herman Venter. 2011. Specification and Verification: The Spec# Experience. *Commun. ACM* 54, 6 (2011), 81–91.

Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. 2005. The Spec# Programming System: An Overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*. 49–69.

Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. 2002. Korat: Automated Testing Based on Java Predicates. In *International Symposium on Software Testing and Analysis*. 123–133.

Eric Bruneton, Romain Lenglet, and Thierry Coupaye. 2002. ASM: A Code Manipulation Tool to Implement Adaptable Systems. In *Adaptable and Extensible Component Systems*.

Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. 2005. An Overview of JML Tools and Applications. *International Journal on Software Tools for Technology Transfer* 7, 3 (2005), 212–232.

Patrice Chalin, Joseph R Kiniry, Gary T Leavens, and Erik Poll. 2005. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In *International Symposium on Formal Methods for Components and Objects*. 342–363.

Patrice Chalin, Robby, Perry R. James, Jooyong Lee, and George Karabotsos. 2010. Towards an Industrial Grade IVE for Java and Next Generation Research Platform for JML. *International Journal on Software Tools for Technology Transfer* 12, 6 (2010), 429–446.

Koen Claessen, Jonas Duregård, and Michal H Palka. 2014. Generating Constrained Random Data with Uniform Distribution. In *International Symposium on Functional and Logic Programming*, Vol. 8475. 18–34.

David R. Cok. 2011. OpenJML: JML for Java 7 by Extending OpenJDK. In *NASA Formal Methods Symposium*. 472–479.

Stephen A. Cook. 1971. The Complexity of Theorem-Proving Procedures. In *Symposium on Theory of Computing*. 151–158.

Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. 2007. Automated Testing of Refactoring Engines. In *Symposium on the Foundations of Software Engineering*. 185–194.

Manuel Fähndrich, Michael Barnett, Daan Leijen, and Francesco Logozzo. 2012. Integrating a Set of Contract Checking Tools into Visual Studio. In *Workshop on Developing Tools as Plug-ins*. 43–48.

Manuel Fähndrich, Michael Barnett, and Francesco Logozzo. 2010. Embedded Contract Languages. In *Symposium on Applied Computing*. 2103–2110.

Burke Fetscher, Koen Claessen, Michał Pałka, John Hughes, and Robert Bruce Findler. 2015. Making random judgments: Automatically generating well-typed terms from the definition of a type-system. In *European Symposium on Programming Languages and Systems*. 383–405.

Norbert E. Fuchs. 1992. Specifications Are (Preferably) Executable. *Software Engineering Journal* 7, 5 (1992), 323–334.

Juan Pablo Galeotti, Nicolás Rosner, Carlos Gustavo López Pombo, and Marcelo Fabian Frias. 2010. Analysis of Invariants for Efficient Bounded Verification. In *International Symposium on Software Testing and Analysis*. 25–36.

Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. 2010. Test Generation through Programming in UDITA. In *International Conference on Software Engineering*. 225–234.

C.A.R. Hoare. 1987. An Overview of Some Formal Methods for Program Design. *Computer* 9 (1987), 85–91.

Daniel Jackson. 2002. Alloy: A Lightweight Object Modelling Notation. *ACM Trans. Softw. Eng. Methodol.* 11, 2 (2002), 256–290.

Eugene Kuleshov. 2007. Using the ASM Framework to Implement Common Java Bytecode Transformation Patterns. In *Aspect-Oriented Software Development*.

Viktor Kuncak, Etienne Kneuss, and Philippe Suter. 2013. Executing Specifications Using Synthesis and Constraint Solving. In *International Conference on Runtime Verification*. 1–20.

Ivan Kuraj, Viktor Kuncak, and Daniel Jackson. 2015. Programming with Enumerable Sets of Structures. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 37–56.

Leonidas Lampropoulos, Diane Gallois-Wong, Cătălin Hriţcu, John Hughes, Benjamin C Pierce, and Li-yao Xia. 2017. Beginner's luck: a language for property-based generators. In *Symposium on Principles of Programming Languages*. 114–129.

Leonid Anatolevich Levin. 1973. Universal Sequential Search Problems. *Problemy Peredachi Informatsii* 9, 3 (1973), 115–116.

Barbara Liskov and John Guttag. 2000. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design.*

Francesco Logozzo. 2013. Practical Specification and Verification with Code Contracts. In *SIGAda Annual Conference on High Integrity Language Technology*. 7–8.

Gus Lopez, Bjørn N. Freeman-Benson, and Alan Borning. 1993. Kaleidoscope: A Constraint Imperative Programming Language. In *Constraint Programming, Proceedings of the NATO Advanced Study Institute on Constraint Programming*. 313–329.

Darko Marinov and Sarfraz Khurshid. 2001. TestEra: A Novel Framework for Automated Testing of Java Programs. In *International Conference on Automated Software Engineering*. 22–31.

Baoluo Meng, Andrew Reynolds, Cesare Tinelli, and Clark Barrett. 2017. Relational Constraint Solving in SMT. In *International Conference on Automated Deduction*. 148–165.

Aleksandar Milicevic, Ido Efrati, and Daniel Jackson. 2014. αRby-An Embedding of Alloy in Ruby. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*. 56–71.

Aleksandar Milicevic, Derek Rayside, Kuat Yessenov, and Daniel Jackson. 2011. Unifying Execution of Imperative and Declarative Code. In *International Conference on Software Engineering*. 511–520.

Joseph P Near and Daniel Jackson. 2010. An Imperative Extension to Alloy. In *International Conference on Abstract State Machines, Alloy, B and Z*. 118–131.

Oracle and/or its affiliates. 2020. *Java™ Platform, Standard Edition 8 API Specification.* https://docs.oracle.com/javase/8/docs/api/.

Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-directed Random Test Generation. In *International Conference on Software Engineering*. 75–84.

Nadia Polikarpova, Carlo A Furia, and Scott West. 2013. To Run What No One Has Run Before: Executing an Intermediate Verification Language. In *International Conference on Runtime Verification*. 251–268.

Derek Rayside, Aleksandar Milicevic, Kuat Yessenov, Greg Dennis, and Daniel Jackson. 2009. Agile Specifications. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 999–1006.

Derek Rayside, Vajihollah Montaghami, Francesca Leung, Albert Yuen, Kevin Xu, and Daniel Jackson. 2012. Synthesizing Iterators from Abstraction Functions. In *International Conference on Generative Programming and Component Engineering*. 31–40.

Christoph Reichenbach, Yannis Smaragdakis, and Neil Immerman. 2012. PQL: A Purely-Declarative Java Extension for Parallel Programming. In *European Conference on Object-Oriented Programming*. 53–78.

Nicolás Rosner, Valeria S. Bengolea, Pablo Ponzio, Shadi Abdul Khalek, Nazareno Aguirre, Marcelo F. Frias, and Sarfraz Khurshid. 2014. Bounded Exhaustive Test Input Generation from Hybrid Invariants. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 655–674.

Hesam Samimi, Ei Darli Aung, and Todd Millstein. 2010. Falling Back on Executable Specifications. In *European Conference on Object-Oriented Programming*. 552–576.

Hesam Samimi, Rebecca Hicks, Ari Fogel, and Todd Millstein. 2013. Declarative mocking. In *International Symposium on Software Testing and Analysis*. 246–256.

Rohan Sharma, Milos Gligoric, Andrea Arcuri, Gordon Fraser, and Darko Marinov. 2011a. *Predicate Coverage.* http://mir.cs.illinois.edu/coverage/.

Rohan Sharma, Milos Gligoric, Andrea Arcuri, Gordon Fraser, and Darko Marinov. 2011b. Testing Container Classes: Random or Systematic?. In *Fundamental Approaches to Software Engineering*. 262–277.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. 404–415.

Emina Torlak and Rastislav Bodik. 2013. Growing Solver-Aided Languages with Rosette. In *Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. 135–152.

Emina Torlak and Daniel Jackson. 2007. Kodkod: A Relational Model Finder. In *Tools and Algorithms for the Construction and Analysis of Systems*. 632–647.

Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. 2012. Green: Reducing, Reusing and Recycling Constraints in Program Analysis. In *International Symposium on the Foundations of Software Engineering*. 58.

Willem Visser, Corina S Păsăreanu, and Radek Pelánek. 2006. Test Input Generation for Java Containers using State Matching. In *International Symposium on Software Testing and Analysis*. 37–48.

Wikipedia. 2020. *Eight Queens Puzzle.* https://en.wikipedia.org/wiki/Eight_queens_puzzle.

Kuat T. Yessenov. 2009a. *JForge: Eclipse Plug-in for Bounded Code Verification.* https://groups.csail.mit.edu/sdg/forge/plugin.html.

Kuat T. Yessenov. 2009b. *A Lightweight Specification Language for Bounded Program Verification.* Ph.D. Dissertation. Massachusetts Institute of Technology.