Side Effect Monitoring for Java using Bytecode Rewriting

Manuel Geffken Peter Thiemann
University of Freiburg, Germany
{geffken,thiemann}@informatik.uni-freiburg.de

Abstract

A side effect of a method in Java is a read or write operation that the method may perform on an object in the heap. Methods with side effects are more difficult to understand and to reason about than pure methods, in particular in the presence of aliasing. While both, Java and the underlying Java Virtual Machine (JVM), support specifying and checking types statically, neither supports ways of specifying and checking side effects statically or at run time.

We propose a technique to specify the side effects of a method and to perform fine-grained monitoring of its side effects at run time. It relies on a domain specific contract language to specify sets of memory locations, assign read and write permissions to them, and enforce these permissions during portions of a program's execution.

We present a specification and an implementation of our monitoring technique.The implementation works independently of any particular JVM by employing bytecode rewriting. The implementation ensures complete interposition while supporting all languages and language features that run on a standard JVM. We evaluate the implementation on a set of benchmarks.

Categories and Subject Descriptors D.2.4 [*SOFTWARE ENGINEERING*]: Software/Program Verification—Assertion checkers, Programming by Contract; D.3.1 [*PROGRAMMING LANGUAGES*]: Formal Definitions and Theory—Semantics

General Terms Verification, Languages

Keywords side effects, contracts, Java, runtime verification, assertion checkers, semantics

1. Introduction

Reasoning about programs with side effects is hard. Despite major advances, the turnkey verification of large programs is still an elusive goal. Therefore, practitioners and researchers advocate testing and monitoring to validate systems and restrict verification to safety-critical parts.

A major example for the monitoring approach is the Design by Contract methodology [27], which roughly consists of having the programmer state assertions for the pre- and postconditions of a method and monitoring their outcome at run time. Pre- and postconditions have been the cornerstones of program verification since its inception [10]. The additional aspect of run-time monitoring has lead to wide-spread adoption of contracts, so that contract monitoring libraries are available for a wide range of languages [8, 9, 18].

Contracts are also the basis of JML, the Java Modeling Language [3, 20], a language for embedding specifications in Java programs. The JML toolchain supports verification as well as runtime monitoring [4, 21, 22]. A JML specification may contain an `assignable` clause to specify the side effects of a method. However, `assignable` is rarely used because its semantics has not been formally and unanimously defined until recently [22]. Furthermore, only a few tools implement run-time monitoring for JML with support for `assignable` clauses [23] and often not in full generality [4]. Lastly, the `assignable` mechanism is less flexible as access permission contracts, because it is bound to fixed classes. We elaborate further on the comparison to JML in Section 6.1.

Our approach has similar goals as the `assignable` clause, but at a finer granularity. We illustrate our approach with the example code in Figure 1. It implements a binary tree data structure with a number of methods that access data in the tree in various ways. Each method is annotated with a contract in a `@Grant` annotation which describes the side effects of the method precisely.

```
class TreeNode {
  static double G = 6.674E-11;

  double mass;
  TreeNode left, right;

  @Grant("this.mass")
  void setMass (double mass) {
    this.mass = mass;
  }

  @Grant("this.mass.@")
  double getMass () {
    return mass;
  }

  @Grant("this.mass.@")
  double mutualForce(
  @Grant("mass.@") TreeNode node,
  double dist) {
    return G * mass * node.mass / (dist * dist);
  }

  @Grant("this.(left|right).@")
  boolean isLeaf() {
    return left == null && right == null;
  }

  @Grant("this.(left|right)*.mass.@")
  double totalMass () {
    return mass + left.totalMass() + right.totalMass();
  }
}
```

Figure 1. A tree node class.

PPPJ '14, September 23–26, 2014, Cracow, Poland..

ACM 978-1-4503-2926-2/14/09. . . $15.00.
http://dx.doi.org/10.1145/2647508.2647515

Each contract consists of an *anchor* and a *path expression*, which together specify a set of memory locations and associate read and write permissions with them. In this context, a location is a pair of an object reference and a field name.

The `setMass` and `getMass` methods are standard setter and getter methods. The setter's contract `this.mass` specifies the method's receiver, `this`, as the anchor and grants read and write permission for the `mass` field. The getter's contract `this.mass.@` only grants read permission, indicated by the trailing `@`.

Contracts may also be attached to method arguments as demonstrated with the first argument of `mutualForce`. In this case, the argument serves as the anchor and the `@Grant` annotation only contains the path expression `mass.@`. Thus, the body of `mutualForce` may only read `node.mass`, but not write to it.

The contract `@Grant("this.(left|right).@")` for `isLeaf` grants read permission for the fields `this.left` and `this.right`. The contract on the `totalMass` method indicates that it recursively reads the `left` and `right` fields of the objects reachable from `this`. At each step, the method may also read the `mass` field, but it must not perform any write operation.

In general, a path expression is a regular expression over the alphabet of field names. An access path has write permission if it fully matches the path expression. An access path has read permission if it is a prefix of a full match.

Our system enforces such contracts at run time for sequential programs (i.e., we do not consider threads). It only permits side effects as specified by the contract. Any unspecified side effect stops the program with an exception.

1.1 Contributions

- *A domain specific language (DSL) to specify (read and write) access permissions on an object graph.* The DSL is based on access paths that start from `this`, a parameter, or a static field.
- *A location-based semantics for this DSL.* A formal specification of the semantics of the DSL. It consists of two parts, a set of locations (object, field name pairs) associated with permissions and the dynamic monitoring of the permissions.
- *An implementation* of the dynamic monitoring that guarantees 100% interposition.* Monitoring is implemented by bytecode rewriting using the Javassist system [5].
- *Evaluation of the implementation.* We present performance data for microbenchmarks and for the Ashes Suite Collection [7].

1.2 Outline

Section 2 presents the underlying semantic principles of our approach and gives an intuitive overview of the implementation. Section 3 defines the syntax and semantics of access permission contracts and their enforcement. Section 4 reports on the implementation of our system in general terms, and then gives details on the Javassist-based implementation with bytecode rewriting. Section 5 evaluates our design with some microbenchmarks and a set of bigger standard benchmarks. Section 6 discusses related work and Section 7 concludes.

2. Monitoring Side Effects

The example from the introduction already provides some intuition, but it leaves open a number of subtle choices that arise in monitoring side effects. This section first discusses and explains these choices. Then it explains the mechanics of monitoring in general terms. Subsequent sections provide further details.

*See `https://github.com/geffken/X2Traverse`.

```
1  double swap (
2  @Grant ("left") TreeNode n1,
3  @Grant ("right, right.mass.@") TreeNode n2) {
4    TreeNode temp = n1.left;
5    n1.left = n2.right;
6    n2.right = temp;
7    return n1.left.mass;
8  }
```

Listing 1. The swap method.

2.1 Semantic Principles

Contracts may be attached to all kinds of methods including constructors. A contract may be anchored at the method's receiver, `this`, or at a static field of a class. A contract in a parameter annotation is implicitly anchored at the corresponding formal parameter. This choice simplifies the implementation: Otherwise, either parameters would have to be addressed by a positional notation, or a precompilation step would be needed to resolve references to parameters by name.

Location-based semantics. Calling a method with a contract evaluates the permissions of the contract and attaches them to locations in the pre-state of the method (i.e., the object graph before executing the method body).

As an example, consider the method `swap` in Listing 1. The contracts on the arguments grant read and write permission to `n1.left` and to `n2.right`. They grant read permission to `n2.right.mass`. Superfically, it might appear that `n1.left.mass` violates the contract, but after executing the assignments, `n1.left` points to the object that used to be `n2.right` in the pre-state. Hence, line 7 actually reads `n2.right.mass` in the pre-state, which is permitted.

The location-based semantics is the semantics of choice for implementing access control for security-inspired scenarios. For program verification and program understanding, a different, path-based semantics is more appropriate as explained elsewhere [16].

In the presence of aliasing, the location-based semantics behaves subtly different to the path-based one. The following example illustrates the different handling of aliasing.

```
@Grant (''this.mass'')
void addTo (
@Grant(''mass.@'') TreeNode n) {
   this.mass += n.mass;
 }
```

Invoking `tree.addTo(tree)` forces `this` and `n` to be aliased. While `this.mass` has read/write permission, `n.mass` only has read permission. With the path-based semantics, the assignment to `this.mass` would be forbidden because the permission is attached to the access path. With the location-based semantics, both paths denote the same location so they have one common permission.

There are two choices for the permission of a location, either the intersection of the permissions computed along all paths that reach the location or their union. Our implementation assigns the union based on the principle of the least surprise to the programmer: it would be very unintuitive to convey why `tree.addTo(tree)` would fail given the above contracts in the code.

Noninterference. An instrumented program that does not violate any contracts should behave exactly as the corresponding program without contracts.

Dynamic extent. The permissions associated to a method by its contracts are in force for each activation of the method, from method entry to the matching return. The permissions are withdrawn at the matching return. The contract therefore extends to all callees of the method. Permissions accumulate in the sense that subsequent contracts can only impose further restrictions.

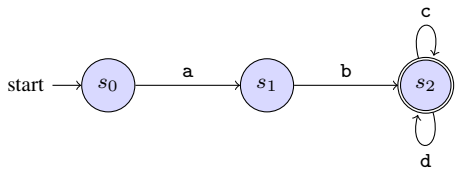

Figure 2. Example automaton for contract `a.b.(c|d)*`.

Pre-state principle. A contract only governs permissions on objects in the pre-state of its method. Objects that are created later on may be read from and written to without restriction, unless a subsequent contract imposes permissions on them.

Default contract. Unannotated methods carry the default contract that enables reading and writing the entire heap. A method or constructor with no externally visible effects may be declared pure by annotating it with an empty contract `@Grant("")`.

Arrays. Arrays require special treatment because they are Java objects without a corresponding class and without designated fields. Read or write permission with respect to an array is homogeneous: all elements of the array have the same permission.

2.2 Contract Enforcement, a High-Level View

Conceptually, our implementations maintains two stacks that operate in lockstep: a stack of sets of new objects and a stack of sets of permission maps.

```
Stack<Set<Object>> newObjs;

enum Access {READ, WRITE}
Stack<Map<Object, Map<Field, EnumSet<Access>>> permissions;
```

The top entry of `newObjs` contains the newly allocated objects since the installation of the currently active permission. All stack entries are disjoint. The elements in the top entry of this stack are not governed by previously installed permissions. Hence, the program is free to read from and write to their fields.

Single permissions are represented by an enumeration `Access`, so that a permission set may be represented efficiently by an `EnumSet<Access>`. Instead of introducing an explicit type of locations, permissions are administered by a two-stage map. The top entry of `permissions` contains the currently active permissions. The default value in a map is the empty set of type `EnumSet<Access>`. This default causes locations with unspecified permissions not to be accessible from contract-annotated methods. Consequently, the currently active permission is always more restrictive than the permission below. The objects in the top entry set of `newObjs` are not in the domain of a map on the `permissions` stack.

These stacks implement the dynamic extent principle. Entry to an annotated method computes a new permission map and pushes it on the stack along with an empty set of new objects.

```
public void installContract(Contract c) {
  Map<Object, Map<Field, EnumSet<Access>>> previous =
      permissions.peek();
  Map<Object, Map<Field, EnumSet<Access>>> next =
      calculatePermissions(c, previous);
  permissions.push(next);
  newObjs.push(Collections.emptySet());
}
```

Returning from an annotated method pops the permission map and merges the two topmost new-object sets on the stack. This merge operation retains the invariant that the top of the new-object stack contains all objects allocated after the installation of the contract that is represented by the top of the permission stack.

```
public void uninstallContract() {
  permissions.pop();
  Set<Object> newLocSinceInstallation = newObjs.pop();
  if (!newObjs.isEmpty()) {
    newObjs.peek().addAll(newLocSinceInstallation);
  }
}
```

Computing the access permission for a given object's field amounts to checking if the object is new, in which case reading and writing is permitted, and otherwise looking up the permission in the top entry of the `permissions` stack. An exception is raised if the desired operation is not permitted.

The calculation of a permission map, represented by the method `calculatePermissions` in the code, requires some preparation at load time. This preparation accesses the `@Grant` annotations, groups the contracts by anchor, and translates each group to a nondeterministic finite automaton. This automaton reads field names and outputs permissions, thus it implements a map that assigns a permission to an access path according to the contract. Figure 2 shows an example automaton.

At run time, each entry to an annotated method starts a heap traversal at the anchor which runs the associated automaton in lockstep. That is, we perform a depth-first traversal of the product graph constructed from the heap and the automaton by matching field names with the automaton's input. To ensure termination the implementation maintains a set of pairs of objects and states to avoid visiting an object twice in the same state.

The heap traversal accumulates permissions in a map P that maps locations to permissions starting from the empty permission map. To visit object ℓ in automaton state q, we first check that this combination has not been seen before. Next, we consider all fields f of object ℓ and build a set of pairs of the form (ℓ', q') such that ℓ' is stored in field f and the automaton steps from q to q' on input f. For each field f in object ℓ, the state q' determines a permission r, which we accumulate in $P(\ell, f)$. The traversal continues to recursively visit the pairs (ℓ', q').

At this point, we have computed a permission map for the new contract. However, further contracts may be installed already and these contracts must not be violated. Thus, we further restrict the permission map by assigning each location the minimum permission from the new map and the already installed permissions. With this map, we finally enter the method body.

3. Specification

This section presents a formal definition of the semantics of the access permission contracts considered in this work. This definition serves as a specification for our implementation.

3.1 Preliminaries

We introduce some notation and basic definitions. A nondeterministic finite automaton (NFA) T is a tuple $(Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ is the input alphabet, $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ is the transition relation, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states. The extended transition relation $\delta^* \subseteq Q \times \Sigma^* \times Q$ induced by this automaton is the least fixpoint of the inclusions (1) and (2) (with $w \in \Sigma^*$ and $x \in \Sigma \cup \{\varepsilon\}$).

$$\delta^* \supseteq \{(q, \varepsilon, q) \mid q \in Q\} \tag{1}$$

$$\delta^* \supseteq \{(q, xw, q'') \mid (q, x, q') \in \delta, (q', w, q'') \in \delta^*\} \tag{2}$$

A state $q \in Q$ is *coaccessible* if there is a string of input symbols $w \in \Sigma^*$ that takes T from q to a final state (i.e., $\exists q_f \in F$: $(q, w, q_f) \in \delta^*$). We write $q \Downarrow$ for "q is coaccessible".

Syntactic domains			
classes	A, B, C	$\in$	$Class$
formal parameters	p	$\in$	$Param$
anchors	x	$\in$	$\texttt{this} + Param + Class$
fields	f	$\in$	$Field$
path expressions	e	$::=$	$\varepsilon \mid f \mid e.e \mid e \mid e \mid e*$
access perm. contr.	c	$::=$	$x.e$
types	τ	$::=$	$prim \mid A \mid \tau[]$
Semantic domains			
primitive values			$Prim$
object references	ℓ	$\in$	Obj
values	Val	$=$	$Obj + Prim$
heaps	H	$\in$	$Obj \times Field \rightharpoonup Val$
access paths	π	$\in$	$Path = Field^+$
path languages	L	$\subseteq$	$Path$
permissions	r	$\in$	$Perm = 2^{\{R,W\}}$
path permissions	M	$\in$	$PathPerm = Path \to Perm$
access permissions	a	$\in$	$Obj \times PathPerm$
permission maps	P	$\in$	$Obj \times Field \rightharpoonup Perm$

Figure 3. Access permission contracts.

3.2 Defining Permissions

Figure 3 defines the syntactic and semantic domains of access permission contracts. Syntactically, an access permission contract $x.e$ is an anchor paired with a path expression. An anchor x may be `this`, a parameter, or a class name. A path expression e is a regular expression over field names: it may be empty, a single field name, a concatenation or an alternative of path expressions, or the Kleene star of a path expression.

In the first step, we resolve the anchor x to an anchor object ℓ_x by an unspecified environment lookup and the path expression to a path permission M, which is a total function from paths to permissions. This function assigns a permission $r \subseteq \{R, W\}$ to each access path π starting from the anchor object. The intended semantics of (ℓ_x, M) is that read access to $\ell.\pi$ is granted if $R \in M(\pi)$ and write access is granted if $W \in M(\pi)$.

We define the path permission $M = M_e \subseteq Path \times \{R, W\}$ for e as the least relation satisfying the following inclusions, where $\mathcal{L}(e)$ is the regular language defined by e.

$$M_e \supseteq \{(\pi, W) \mid \pi \in \mathcal{L}(e)\} \quad (3)$$

$$M_e \supseteq \{(\pi, R) \mid (\pi, W) \in M_e\} \quad (4)$$

$$M_e \supseteq \{(\pi, R) \mid (\pi.f, R) \in M_e\} \quad (5)$$

Inclusion (3) states that each full path $\pi \in \mathcal{L}(e)$ is writeable; inclusion (4) states that each writeable path is also readable; and inclusion (5) closes readability under prefixes. The read-only marker `@` does not require special treatment: we regard it as a special field name that does not occur in any program. We consider M_e as a set-valued total function in the usual way. Section 3.3 shows that the function M_e may be computed by a nondeterministic finite automaton because the preimages of R and W are both regular languages.

Having obtained a path permission M_e in this way, the semantics of an access permission (ℓ_x, M_e) is defined relative to a heap H, which is modeled as a partial function from locations (object addresses and field names) to values. The definition of the semantics has two stages. The first stage defines a set $R(H, \ell_x)$ consisting of all triples $(\ell, f, \pi.f) \in Obj \times Field \times Path$ such that object ℓ is reachable from the anchor ℓ_x via path π and f is a field in the object at ℓ. It is defined as the least fixpoint of two inclusions.

$$R(H, \ell_x) \supseteq \{(\ell_x, f, f) \mid (\ell_x, f) \in dom(H)\} \quad (6)$$

$$R(H, \ell_x) \supseteq \{(\ell, f, \pi.f) \mid (\ell', f', \pi) \in R(H, \ell_x), \ \ell = H(\ell', f'), (\ell, f) \in dom(H)\} \quad (7)$$

Inclusion (6) specifies the initial path segments starting from x. Inclusion (7) extends a path segment if a reachable object at ℓ' contains an object $\ell \in Obj$ in field f'. By construction, the last step of the path component is always equal to the field component in the middle.

Due to aliasing, more than one path may reach the same object ℓ. Our design computes the permission for field f of object ℓ as the union of the permissions for all paths that reach $\ell.f$.

$$P(\ell, f) = \bigcup \{M(\pi) \mid (\ell, f, \pi) \in R(H, \ell_x)\} \quad (8)$$

As the specification (8) quantifies over a potentially infinite set of paths, we need an algorithm to compute $P(\ell, f)$ in finite time. Section 3.3 specifies this algorithm.

3.3 Computing Path Permissions

Our algorithmic representation of a path permission M_e is an NFA $T_e = \mathrm{nfa}(e)$, which we construct from a path expression e using the standard construction. To compute the permission for a path, it suffices to run T_e on the path and check the resulting set of states for coaccessible and final states.

Concretely, if $T_e = (Q, Field, \delta, q_0, F)$, then define

$$M_e^{NFA}(\pi) = \bigcup \{r(q) \mid (q_0, \pi, q) \in \delta^*\} \quad (9)$$

$$r(q) = \{R \mid q \Downarrow\} \cup \{W \mid q \in F\} \quad (10)$$

where $\pi \in Field^*$ and δ^* is the extended transition relation of T_e.

Lemma 1 *For all path expressions e and paths π, $M_e(\pi) = M_e^{NFA}(\pi)$.*

PROOF We first establish the following correspondences by induction on e using well-known results about the composition operators $\cdot$ (concatenation), $\cup$ (union), and * (Kleene star) for NFAs.

Let δ^* be the extended transition relation of T_e.

$$\pi \in \mathcal{L}(e) \Leftrightarrow \exists q' \in F.(q_0, \pi, q') \in \delta^* \quad (11)$$

$$\pi.\pi' \in \mathcal{L}(e) \Leftrightarrow \exists q \in Q. \exists q' \in F. \ (q_0, \pi, q) \in \delta^* \wedge (q, \pi', q') \in \delta^* \quad (12)$$

These correspondences enable us to check properties (3), (4), and (5) for $M_e^{NFA}(\pi)$ and to show that it contains no further elements.

In the implementation, we do not use δ^* but maintain the set of states that has been reached by the last step of a path. We further construct a *trim* NFA, where all states are coaccessible, so that checking read permissions boils down to checking whether the current state set is non-empty. This check is needed anyway to stop the heap traversal at the boundary of the path language where no further access-permitted locations can be found.

The function M'_e captures the result of our improvements. It directly reflects our implementation.

$$M'_e(\varepsilon) = (\emptyset, \{q' \mid (q_0, \varepsilon, q') \in \delta_e^*\}) \quad (13)$$

$$M'_e(\pi.f) = (r, Q') \text{ where} \quad (14)$$

$$(r_\pi, Q_\pi) = M'_e(\pi)$$

$$Q' = \{q' \mid (q, f, q') \in \delta_e^*, q \in Q_\pi\}$$

$$r = \bigcup \{Perm(q') \mid q' \in Q'\}$$

$$Perm(q') = \{R\} \cup \{W \mid q' \in F\}$$

In the second component, the function M' computes the current states of the NFA. In the first component, it computes the union of all permissions according to the current states. Each state represents a set of paths that grants the same permission. As the underlying NFA is trim, δ_e^* only yields coaccessible states q'. For the empty path, the first component is empty and the second component contains all coaccessible states that are reachable from the initial state q_0 without consuming any input. For a path of the form $\pi.f$, the function examines the states Q_π that are reached by $M'_e(\pi)$. It collects the successor states for all transitions from state $q \in Q_\pi$ into a state q' on symbol f in Q'. From Q' it computes the permission by applying $Perm$ pointwise to Q' and taking the union.

Finally, we can prove that the thus computed permission map M'_e also adheres to its specification M_e. Besides Lemma 1 we need two more lemmas to establish the correspondance.

Lemma 2 *If $M'_e(\pi) = (r, Q')$, then $Q' = \{q' \mid (q_0, \pi, q') \in \delta_e^*\}$.*

Lemma 3 *For all π, $M_e^{NFA}(\pi) = r$ iff exists Q' such that $M'_e(\pi) = (r, Q')$.*

4. Implementation

This section first explains general implementation considerations regarding the choice of data structures. Next, we discuss the implementation of the NFAs generated from contracts. Finally, we give details from our bytecode rewriting implementation using Javassist.

4.1 Data Structures

Section 2.2 explains that two stacks, `newObjs` and `permissions`, containing sets and maps are needed to represent the current state of the permission monitoring system. It turns out that these data structures have special requirements.

First, the objects stored in the sets and maps are arbitrary user objects. For correctness, all lookup operations in these sets and maps must compare objects using reference equality (identity). However, this requirement rules out using Java's standard collection classes, as they rely on the `equals` and `hashCode` methods. These methods may be overridden by the user program and thus yield unpredictable or even incorrect results. Therefore, we use an implementation along the lines of Java's `IdentityHashMap`, which uses object identity and maintains the constant time cost of the lookup operation on field accesses.

However, we cannot use `IdentityHashMap` directly because a permission map may refer to objects that are no longer referenced by the application program. Hence, these sets and maps should only *weakly* refer to objects so that they can be garbage collected if no other reference exists. Unfortunately, the Java standard library does not contain a weak version of `IdentityHashMap`. Thus, our implementation includes a `WeakIdentityHashMap`, which is an adaptation of `WeakHashMap` to use the identity hash function.[†]

In our first attempt at implementing permission contracts, the `newObjs` stack was a major bottleneck. However, it turns out that this stack can be elided. The idea is to establish the relative order of both installed contracts and objects by introducing *contract generations*. On contract installation, we assign a contract generation that is larger than all previous contract generations. Whenever the program creates a new object, it attaches the current generation to the object. To enforce the `newObjs` stack discipline, we postulate that an object is only governed by a contract if the contract's generation is greater than the object's generation. Otherwise, it is new and thus unconstrained.

[†]The `MapMaker` class from the Google guava library provides similar functionality, but is based on a *concurrent* map implementation, which has a higher overhead. See `http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/collect/MapMaker.html`

```
WeakIdentityHashMap<Object, Long> objGenMap;
Stack<Long> objGens;

Stack<WeakIdentityHashMap<Object, Map<Field, EnumSet<Access>>>
    permissions;
```

Listing 2. Revised data structures for access contract monitoring.

We implement contract generations as a weak identity hash map `objGenMap` that maps an object to its contract generation and a stack of contract generations `objGens` to represent the `newObjs` stack from Section 2.2. This representation avoids the expensive set merge operation on contract uninstallation. The check if an object is new has roughly the same cost. In a naive implementation of the `newObjs` stack checking if an object is new amounts to determining whether the object is an element of the top entry set of the `newObjs` stack. Using generations this check amounts to looking up the object's generation in the `objGenMap` (which has comparable costs as the original "contains" check) and comparing the object's generation with the currently installed contract's generation.

Listing 2 contains the revised declarations for the data structures. They fulfill all requirements and enable a reasonably efficient implementation of access contracts.

4.2 Contract Installation and Uninstallation

Access permission contracts are embedded in Java in the form of annotations (Java Language Specification [12], Sec. 9.7). These annotations need not to be accessed at run time, that is, their retention policy is `CLASS`. Each access permission annotation $x.e$ is translated to an anchor ℓ_x and an NFA T_e that implements the permission mapping M from Section 3.2. Section 3.3 details the translation of a path expression e to its corresponding NFA T_e. To avoid simulating the nondeterministic automaton at run time, the implementation applies the standard powerset construction to create an equivalent deterministic automaton D_e before installing it. The transition function δ_e of this automaton is δ_e^* defined according to (1) and (2):

$$\delta_e(\bar{q}, f) = \{q' \in Q \mid q \in \bar{q}, (q, f, q') \in \delta_e^*\}$$

and its initial state is

$$\bar{q}_0 = \{q \mid (q_0, \varepsilon, q) \in \delta_e^*\}.$$

As the states of the deterministic automaton D_e are sets of states of T_e, it is easy to come up with an efficient encoding of sets of states by numbers, which reduces the implementation of δ_e to an array of maps (from field names to states) and the initial state to a single number. The final states may be represented by negative numbers, which is cheaper to check than membership in a set. Unfortunately, we are forced to keep a string representation of field names because the definition of a contract and its translation to an automaton is independent of any class. Furthermore, the automaton must be usable with any number of classes defined in the future. Clearly, we cannot hope to find an encoding of field names as numbers with unlimited extensibility. Thus, we obtain the following encoding of an automaton.

```
class DFA {
  Map<Field, Integer> [] delta;
  int initialState;
}
```

The assignment of read and write permissions requires implementing the permission computation shown in (6), (7), and (8) by a heap traversal. Calling an annotated method starts a heap traversal at the anchor which runs the automaton D_e alongside as specified

by the function M'_e in (13). The initial state for the traversal is the initial state of D_e, $\bar{q}_0$.

The traversal procedure takes an object ℓ and a DFA state $\bar{q}$ as arguments. It returns immediately if this combination has been seen before. Otherwise, it considers all fields f of object ℓ and builds a set of pairs that contains each object ℓ'_f that is accessible with a single field access on f from ℓ paired with its corresponding DFA state $\bar{q}'_f = \delta_e(\bar{q}, f)$ according to the transition representing the traversed field from the current state $\bar{q}$. As the automaton only contains coaccessible states, the transition may not be defined for some fields. For each field f with a valid transition, the traversal code accumulates the output permission in a global map $P(\ell, f)$ (cf. (8)). Then the traversal continues by recursively visiting the new pairs $(\ell'_f, \bar{q}'_f)$ until no unvisited pairs remain.

When leaving the method that installed the new contract, it need to revert to the previously installed permissions. The implementation pops the current permission.

4.3 Javassist Implementation Specifics

Javassist [5] is a Java bytecode manipulation library that provides instrumentation both at compile time and at load time (structural reflection). It enables Java programs to define a new class at run time and to modify a class file when the JVM loads it. Javassist is independent of the particular JVM. It provides a source-level API that enables code manipulation without the need to dig into the peculiarities of Java bytecode. Bytecode snippets to be inserted may be specified in Java source code augmented with some template operations.

4.3.1 Heap Traversal

Our implementation analyzes each class that needs to be traversed and generates specialized traversal code for objects of that class. Using this technique avoids the expensive run-time reflection to determine what fields exist in an object during the heap traversal. The actual traversal is implemented using the visitor pattern. The visitor runs the automaton representation of a contract on all fields in an object and assigns permissions to locations. The pseudocode for the `visit` method is as follows.

```
void visit(Object obj, String fieldname, Object fieldvalue) {
  if (!(fieldvalue instanceof Traversable)) {
    return;
  }
  // remember automaton state
  // step automaton according to field name
  // calculate effective permission and update permission map
  ((Traversable)fieldvalue).traverse(this);
  // restore remembered automaton state
}
```

To make the heap traversable, each application class gets extended to implement the `Traversable` interface. We implemented a custom classloader that performs this extension at load time using Javassist. The particular implementation of the `traverse` method is generated according to the fields that are declared in the class. As an illustrating example, we consider the following class `C` that contains one field of reference type and one primitive field.

```
public class C extends D {
  F f;
  int g;
}
```

The `traverse` method to be generated for class `C` is the following.

```
public void traverse(Traversal t) {
    t.visit(this, "f", this.f);
    t.visitPrimitive(this, "g");
    super.traverse(t);
}
```

We call the visitor's `visit` method to compute the permission for location `this.f` and initiate the traversal from the value of `this.f`. For the primitive field `g`, we only need to compute the location's permission by calling `visitPrimitive`.

4.3.2 Bytecode Instrumentation

The instrumentation of field access and object creation in a method body can be achieved by means of Javassist's expression editor. Field access expressions of all methods and constructors of a target class are instrumented as outlined below.

```
methodOrCtor.instrument(new ExprEditor() {
  public void edit(FieldAccess expr) {
    StringBuilder code = new StringBuilder();
    code.append("{");

    // get active permission for location to access
    code.append("EnumSet<Access> effectivePerm = getPermission($0, \""
                            + expr.getFieldName + "\");");

    // get permission needed for this access
    code.append("Access access = Access."
        + (expr.isReader() ? "READ" : "WRITE") + ";");

    code.append("if (!effectivePerm.contains(access)) {");
    if (expr.isReader()) {
      code.append(" throw new ReadViolation();");
    } else {
      code.append(" throw new WriteViolation();");
    }
    code.append("}");

    if (expr.isReader()) {
      code.append(" $_ = $proceed();");
    } else {
      code.append(" $proceed($$);");
    }
    code.append("}");
    expr.replace(code.toString());
  }
});
```

We create a code template that is to be compiled to an instruction sequence that replaces the original field access. In this code fragment, the statement `$_ = $proceed();` stands for a read access, `$proceed($$);` represents a write access, and `$0` is the object containing the field. We insert the instrumentation enforcing the access permissions before the field access.

First, we compute the effective permission for the location to access by calling `getPermission`. Then, for a read access, we check whether R is contained in the effective permission. Likewise, for a write access, we check for W. If the access is not permitted, we throw the appropriate access violation exception.

Instrumentation of object creation is more involved than field access instrumentation. The instrumentation needs to be inserted after the object has been allocated via the bytecode but before the execution of the constructor body which may contain effects to be monitored. Moreover, the instrumentation code must not store the new object on the `newObjs` stack before its initialization is complete. Hence, we insert the code after the superclass constructor has been called. We achieve this instrumentation using Javassist's expression editor to edit the (`super` or `this`) constructor call within the constructor body.

```
ctor.instrument(new ExprEditor() {
  public void edit(ConstructorCall expr) {
    StringBuilder code = new StringBuilder();
    code.append("{");
    code.append(" $_ = $proceed($$);");
    code.append(" if (!objGens.isEmpty())")
    code.append(" objGenMap.put($0, objGens.peek());");
    code.append("}");
    expr.replace(code.toString());
  }
});
```

In the replacement code for the constructor call, the statement `$_ = $proceed($$);` inserts the original constructor call and `$0` represents the object under construction, to which we assign the most recently installed contract's generation.

4.3.3 Static Fields

Static fields are the second important class of roots into a method's visible heap. In contrast to method parameters, which are located on the stack, static fields are heap locations themselves. Thus, we need to maintain permissions for the static fields and the locations that can be reached from them. Thus, we need to generate specialized code that calculates the permissions for all static fields and initiates a heap traversal from static fields of reference type.

It turns out that static initialization makes it very hard to maintain noninterference if contracts mention static fields. To guarantee noninterference, we must make sure that static initializer blocks and field initializers for class variables run at the same time in the instrumented program as in the uninstrumented program. In particular, the instrumentation should not trigger class loading.

However, if the instrumentation is not allowed to trigger class loading, then we cannot grant all permissions given by a contract because granting permission requires to traverse the heap starting from the static fields for "deep" access paths rooted in static fields, which in turn triggers initialization. In this scenario, class-anchored permissions could only be granted for locations in objects of already initialized classes and for the static fields themselves because the heap structure does not influence the interpretation of access paths to such fields but is determined statically. If we granted these permissions "lazily" on class initialization, then we would not meet the pre-state principle because the heap structure may have changed arbitrarily by the time initialization takes place compared to the respective pre-state.

Besides the complicated implementation, we believe that the pre-state principle is more important than order of initialization. Hence, our solution is to go for a weaker guarantee: An instrumented program that does not violate any contracts behaves exactly as the uninstrumented program *if the program's behavior does not depend on the order in which classes are initialized, as long as the constraints imposed by Java are respected.*

We believe that a reasonable Java programs meets this condition because a program violating this rule does add non-obvious global dependencies across the program and does therefore not impose additional restrictions on the programming style in most cases. The condition enables us to prepone static initialization in the instrumented program and grant all permissions given by a contract regardless of the initialization state of the program while adhering to the pre-state principle.

Do contracts apply to static initializers? Generally, a contract applies to all class initializers that are executed while the contract is installed as it applies to all other executed code. However, we do not enforce any restrictions on static fields that are accessed by the declaring class's initializer. At that time, we consider these locations as "new" locations.

4.3.4 Arrays and Object

Arrays also lead to subtle difficulties for the traversal. In particular, we cannot instrument arrays with a `traverse` method because there is no classfile representation for array types where our specialized code could be entered. Moreover, there is no common superclass for arrays apart from `Object`, to which we could add a `traverse` method. We chose not to instrument the `Object` class because `Object` should not implement any interface. Hence, we need to generate explicit traversal code for an array-typed field in the visitor of the class containing this field.

However, there is a further catch. A field of type `Object` may contain an array and an array with base type `Object` may contain elements that are arrays. In such cases, we generate code to get down to the `Object`-typed values. For each value, the code first tests whether the value implements `Traversable`. If it does, the generated code invokes `accept` to continue. Otherwise, the code checks whether the dynamic type of the value is an array, traverses this array, and checks its elements recursively.

Fortunately, the information required to statically generate as much of the traversal code for arrays as possible is available from the static types. The following inference rules reflect the reasoning that is needed. The first set of rules covers those cases where we can infer from an expression's static type that an expression does not have type "reference array" and can therefore omit a dynamic type check. In fact, for T-PRIM and T-PRIM-ARR no traversal code needs to be generated.

$$\text{T-OBJ}\quad \frac{\Gamma \vdash p : A \qquad A \neq Object \qquad \forall \tau.\tau[] \neq A}{\Gamma \not\vdash p : Object[]}$$

$$\text{T-PRIM}\quad \frac{\Gamma \vdash p : prim}{\Gamma \not\vdash p : Object[]} \qquad \text{T-PRIM-ARR}\quad \frac{\Gamma \vdash p : prim[]}{\Gamma \not\vdash p : Object[]}$$

Likewise, exploiting Java's subtyping rules we can identify some static types as *definite* subtypes of $Object[]$, which also avoids a dynamic type check.

$$\text{T-PRIM-MULT-ARR}\quad \frac{\Gamma \vdash p : prim[]^{n+1} \qquad n > 0}{\Gamma \vdash p : Object[]^{n}} \qquad \text{T-MULT-ARR}\quad \frac{\Gamma \vdash p : A[]^{n}}{\Gamma \vdash p : Object[]^{n}}$$

The two rules follow from standard Java subtyping rules. If one of these rules is applicable, we can avoid a dynamic type test for a reference array on up to n levels of heap traversal by generating specialized traversal code that takes this information into account.

5. Evaluation

The first part of the evaluation concentrates on the run-time overhead caused by installing contracts and enforcing the associated permissions. We run all benchmarks with `java` in version "1.7.0_51" on the OpenJDK 64-Bit Server VM (build 24.45-b08, mixed mode) and use the OpenJDK Runtime Environment (IcedTea 2.4.4). First, we consider micro benchmarks, then we consider a set of benchmarks from the Ashes Suite Collection [7].

All benchmarks were executed on a Dell Optiflex 7010 with an Intel Core i5-3570 processor (3.4 GHz, 4 Cores, and 16 GB RAM on top of Linux Mint 16.

5.1 Micro Benchmarks

If not mentioned otherwise, all reported figures are accumulated run times of 100 runs of an individual benchmark. A warm-up phase was executed before the start of the timed benchmark.

While instrumentations of the read, write, and new operations impose a constant run-time overhead, each contract installation involves a heap traversal to assign access permissions to locations. Consequently, we expect calls of contract-annotated methods to be expensive if they impose access permissions on large data structures.

5.1.1 Run Time Performance of Trees and Linked Lists

Our first micro benchmark extends the introductory example class `TreeNode` in Figure 1 according to Listing 3. The benchmark applies the recursive function `computeForces` to two sets of inputs: balanced and degenerate trees (linked lists). We run each input set in two variants, once (marked "Full") with the directly

```
@Grant("this.(left|right)*.force, this.(left|right)*.mass.@")
void computeForces() {
  /* reads mass writes force */
  this.force = this.mass * G;
  /* reads left.(left|right)*.mass, right.(left|right)
     writes left.(left|right)*.force */
  if (left != null) left.computeForces();
  /* reads right.(left|right)*.mass, right.(left|right)
     writes right.(left|right)*.force */
  if (right != null) right.computeForces();
}

@Grant("this.(left|right)*.force, this.(left|right)*.mass.@")
public void computeForcesDelegator() {
  computeForcesUnannotated();
}
```

Listing 3. Variations of `computeForces` in `TreeNode`.

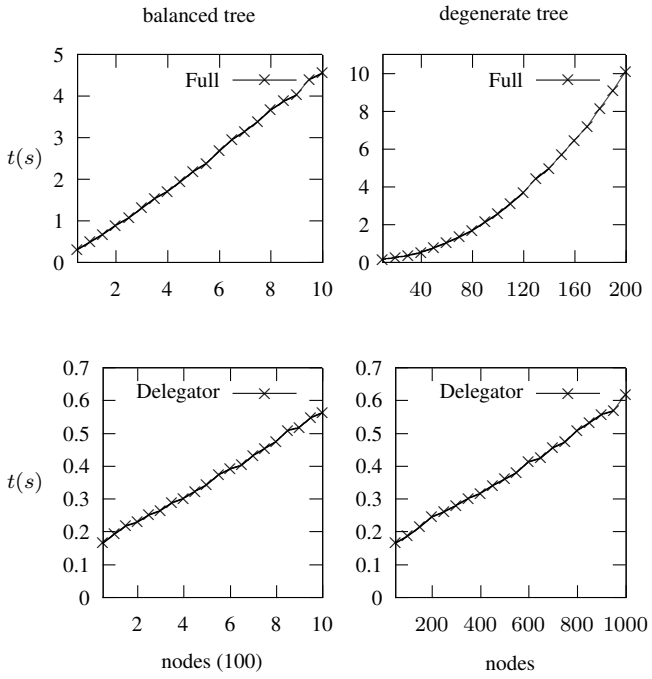

Figure 4. Balanced tree (left) and linked list (right) case study for `computeForces`. X-axis: number of nodes, scaled by a factor of hundred for the left column. Y-axis: time in seconds.

annotated recursive function `computeForces` and once (marked "Delegator") with an intermediate, annotated wrapper method `computeForcesDelegator`. The latter variant avoids multiple recursive installations of the same contract on the permission stack.

The left column of Figure 4 depicts the outcome of our benchmarks for different tree sizes, whereas the right column presents the results for different sizes of linked lists. Sizes are measured in the number of nodes on the x-axis, where run times are measured in seconds on the y-axis. Let us first focus on the "Full" case in our analysis of the graphs at the top of the figure.

The "balanced" benchmark shows that the run time grows proportionally to $n \log n$ were n is the number of tree nodes. This growth can be explained by calculating the accumulated size $d_a(n)$ of the accessible heap for all calls of `computeForces` on a balanced tree with n nodes. Because $d_a(n) = n + 2d_a(n/2)$ it holds that $d_a(n) \in O(n \log n)$ by the Master theorem.

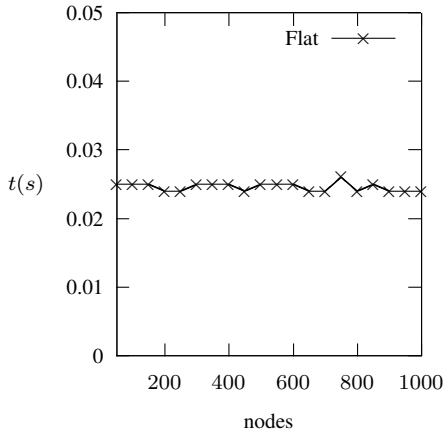

Figure 5. Linked list case study for `computeForce`. X-axis: number of nodes. Y-axis: time in seconds.

The "degenerate" benchmark exhibits a run time that is quadratic in the number of nodes. Again, this growth can be explained by analyzing the accumulated depth $d_a(n)$ of the accessible heap which grows quadratically for a degenerate tree: $d_a(n) = n + d_a(n-1)$.

The partly-annotated "Delegator" variant at the bottom of the figure demonstrates how the structure of our contract language supports an infinite number of locations and helps to improve the scalability of our approach for recursive calls. Furthermore, the partly-annotated variant's performance is much more relevant for idiomatic Java code which often prefers iteration over recursion to avoid overflowing the run-time stack.

A closer look at the lower-left corner of the graphs at the bottom reveals that for small numbers of nodes (less than about 100), the implementation does not scale lineary as it does for higher numbers of nodes. We do not have a good explanation for this behavior, but suspect that it is due to cache effects.

5.1.2 Contracts Covering Few Locations

So far our benchmarks focused on contracts covering the majority of the accessible heap. However, many methods only access a small fraction of the accessible heap. Accessors and mutators are examples for such methods. These methods can be annotated with contracts where the number of locations covered by the contract is small.

To look into this scenario, we analyze a "flat" variant of the method `computeForces` called `computeForce`, which accesses just a single node, and apply it to the root of each input tree.

```
@Grant("this.force, this.mass.@")
public void computeForce() {
  this.force = this.mass * G;
}
```

The method's contract refers to a small constant number of locations which is independent of the size of the accessible heap. As we have already identified degenerate trees as the pathological case, Figure 5 presents the results for such trees. The overhead for contract installation remains constant and is independent of the size of the accessible heap. Given a contract whose access path language covers only part of the accessible heap our implementation stops the heap traversal at the boundary of the path language and implicitly assigns an empty permission to the remaining locations.

5.1.3 Unannotated Methods

The last set of benchmark runs considers programs without annotations and compares their run times when running with instrumenta-

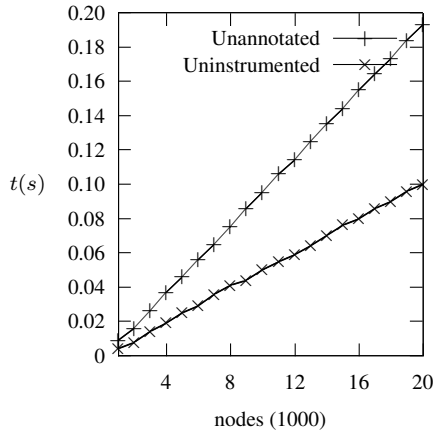

Figure 6. Linked list case study for unannotated and uninstrumented `computeForces`. X-axis: number of nodes in thousands. Y-axis: time in seconds.

tion or without. This study determines the raw overhead of the instrumentation when it is not exercised. The program in this benchmark is exactly the same program used to produce the right column of Figure 4, except that we have scaled the number of nodes up to 20000 to obtain stable measurements. In the benchmarks covered in this section, we increase the number of runs to 1000 to overcome the limited granularity of the used timing mechanisms.

The graph in Figure 6 plots the uninstrumented run time against the instrumented implementation. The instrumented annotation imposes a 1.93x slowdown (0.193s/0.1s). This comparably low overhead can be explained by the fact that we can avoid the method entry and exit instrumentation for unannotated methods altogether. Instead, checking for a contract happens in the initial instrumentation phase only. The run time is roughly linear in the number of nodes, which demonstrates that the overhead at each read, write, and new operation is constant.

As indicated at the beginning of this subsection, the underlying program and input data for Figure 6 is the same as for Figure 4, right column. The run times in Figure 6 are clearly linear in the number of nodes. We want to use these observations to estimate the total overhead of a non-trivial permission that is only installed once per run.

From the raw data for the unannotated Javassist-based version we obtain an average timing of 0.100s for 20000 nodes and 0.193s for the unmonitored version. The corresponding data underlying Figure 4 is 0.618s for 1000 nodes. Conservatively estimating the run time for 1000 runs as 10x higher than for the actual 100 runs, we obtain 6.18s for 1000 nodes. Thus, the overhead of enforcing a non-trivial contract is 1601x and the total overhead against the unmonitored version is 3090x.

While our implementation based on bytecode rewriting imposes a low overhead for the instrumentation of the field access and object creation, the overhead for the contract installation is high for large heaps. We believe that the traversal algorithm is still too generic and that further improvements are possible. In the following section we analyse how our system impacts the run-time performance of bigger, more realistic applications.

5.2 Ashes Benchmark Suite

We analyse a set of benchmark programs from the Ashes Suite Collection. We chose the "ashesHardTestSuite" because the benchmark programs in that suite expose all static ingredients that we are relevant for our instrumentation: static and instance field accesses, array accesses, object creations and method calls. We ignore the effects produced by standard libraries and focus on application code.

We present the results of running the instrumented and uninstrumented programs from the "ashesHardTestSuite" in Table 1. We had to exclude two programs from our measurements, `javazoom` and `puzzle`. We do not present run times for the `javazoom` benchmark because a method's maximal size was exceeded after instrumentation. The increase was from 40176 bytes to 74131 bytes, while the maximum method size is 65534 bytes. We exclude the `puzzle` benchmark from our evaluation because the original (uninstrumented) benchmark did not complete successfully on our system but threw an exception.

For each application, we state the name, the number of classes and methods, different instrumented run times and the uninstrumented ("Base") run time. The column "Unannotated" lists the run times of the instrumented applications without contract annotations (but field accesses and object creations are instrumented). Column "Annotated" lists the run times of the instrumented applications being annotated with default contracts granting full access to all locations. In the "Unannotated" version we distinguish "online" (load time) instrumentation and "offline" instrumentation before the execution of the program. Additionally, we present the number of traversed objects (excluding array objects) and the instrumentation times for the "Annotated" applications.

For example, the method `matmul` in the `matrix` benchmark consisting of the two classes `Matrix` and `Test` carries the following the default contract.

```
@Grant("Matrix.*, Test.*, this.*")
public void matmul(@Grant("*") Matrix a, @Grant("*") Matrix b) {
  ...
}
```

Although the default contracts are synthetic, evaluation of a program annotated with these contracts demonstrates an interesting performance point because such contracts require the largest possible heap traversal. Hence, considering the evaluation results for the micro benchmarks in Section 5.1, where we identify the size of the heap to be traversed as most significant for the imposed run-time overhead, we can interpret the obtained run time of a program annotated with these default contracts as the worst case for the respective program.

On the other hand, the unannotated run time can be considered the best case performance for the instrumented program.

All benchmarks show a relatively small overhead in the unannotated version. Comparing the "Base" version with the "offline"-instrumented "Unannotated" version the overhead ranges between about 4% in the `probe` benchmark and 57% in the `illness` benchmark. The run times of the run-time instrumented ("online") version are higher as they include the time spent in Javassist. This instrumentation time amounts to the listed run time less the run time of the "offline" version. The instrumenation time of the default "Annotated" applications are generally higher than in the "Unannotated" version because additional instrumentation is needed for contract installation and uninstallation but the overhead stays within practical bounds.

Now, we consider the run times of the instrumented "Annotated" benchmarks. While some benchmarks (`fft`, `lu`, `probe`) have a relatively low total overhead (1.66x to 2.66x), other benchmarks (`boyer`, `illness`, `lexgen`) show a very high overhead (13735x to 36051x) against the "Base" version. The `matrix` and `decode` benchmarks show a high overhead (1719x to 4662x). The `machineSim` benchmark runs about 195x slower than the "Base" version.

First, we look at those benchmark that have a low overhead: `fft`, `lu`, and `machineSim`. These benchmarks traverse no (non-array) objects and contract installation und uninstallation has a

Application	Classes	Methods	Run time [s]	Run time [s] of instrumented programs				
			Base	Unannotated		Annotated (offline)		
				online	offline		Traversed objects	Instr. time [s]
boyer	31	75	0.27	1.01	0.35	9733.82	6,695,049,495	1.82
decode	4	19	0.23	0.61	0.32	395.40	18,398,341	0.50
fft	1	10	0.06	0.24	0.08	0.10	0	0.38
illness	17	93	0.14	0.97	0.22	1923.01	159,195,509	1.04
javazoom	32	318	0.85	-	-	-	-	-
lexgen	64	210	0.33	2.47	0.52	7634.90	1,583,107,978	4.07
lu	1	7	0.24	0.42	0.26	0.64	0	0.34
machineSim	11	108	0.12	1.45	0.16	23.42	8,423,971	1.90
matrix	2	10	0.17	0.51	0.26	792.57	0	0.44
probe	1	6	0.70	0.96	0.73	1.26	0	0.35
puzzle	3	20	-	-	-	-	-	-

Table 1. Analysis results for the "ashesHardTestSuite" benchmarks from the Ashes Suite Collection.

very low overhead. Additionally, they contain virtually no field accesses, object creations, or uses of non-primitive arrays beyond the standard library.

Next, we consider `decode` and `matrix`. The run time of the `decode` benchmark can be explained by the high number of objects traversed (18,398,341). For `matrix` we suspect that the run time can be explained by a high number of two dimensional arrays that contain references that need to be traversed. We excluded these array objects from the presented object count as their traversal has different run-time characteristics from the traversal of "normal" objects.

The very high overhead of `boyer`, `illness`, and `lexgen` has one main reason. Both benchmarks create a large number of objects on the heap. In the `boyer` benchmark the heap gradually grows to about 70,000 objects with about two fields on average. For many method calls, these objects are on the visible heap. Thus, at each method call a large number of objects needs to be traversed to install the fully permissive contract. In contrast, the original benchmark accesses only a small fraction of this visible heap in each method. We counted 6,695,049,495 (excluding array objects) objects visited across all traversals. In the `illness` and `lexgen` benchmarks 159,195,509 objects and 1,583,107,978 objects respectively had to be traversed.

Generally, the run time of a realistically annotated program can be assumed to lie between the "Unannotated" and the "Annotated" version. Within this large range, we expect the run time of a program carrying reasonably restrictive contracts to be significantly closer to the best case because realistic contracts would generally only allow access to a small fraction of the accessible heap, which we have identified in Sec. 5.1.2 as a case that imposes a relatively low overhead. We have not annotated the benchmarks with more realistic contracts yet because obtaining such contracts requires either a deep insight into the program and a lot of manual work, or a kind of effect inference, which we do not have in place yet, although this seems technically feasible to us [14, 15].

In the current implementation, although we generate specialized code that initiates the traversal from the static fields of each class, we call this code using reflection. We can avoid these reflective calls by generating specialized code for each contract that calls into the traversal code of all classes being an anchor in the contract. In our experience avoiding reflection can speed up the traversal significantly for contracts containing many class-anchored permissions.

6. Related Work

6.1 Comparison to JML's `assignable` Clause

In terms of specification languages, JML's `assignable` clause has been discussed in the introduction. There are systems that support static verification as well as monitoring of `assignable` clauses [23]. Data groups extend `assignable` clauses similar to an effect system [24]. They can be modeled by our path expressions but are more abstract because they do not mention concrete field names. A similar abstraction could be included in our system, too.

Concretely, an assignable clause may specify a finite set of access pathes starting from `this`, instance variables, or static variables using a store-ref-list. For arrays, it is possible to specify a subset of the indices as assignable. We do not provide this feature currently, but it would be straightforward to add. For objects, it is possible to terminate a store-ref (SR) with `*`, which means that "the union of the data groups of all visible instance fields of SR's (static) type" is also assignable [19, Section 12.7].

The purpose of a data group is structured information hiding in an object-oriented specification [24, 25]: A specification should neither reveal nor rely on the names of instance variables in a class. The simplest data group (as incorporated in JML) consists of a single (instance) variable and is named like the variable. A static data group just collects a set of variables under a new name. A dynamic data group collects ("maps" in JML data group jargon) the members of data groups of objects that are referred to by the current object. If a class contains recursive fields, then such a data group may be arbitrarily large.

Except for array subranges, access permission contracts are able to express all restrictions imposed by assignable clauses with data groups. For finite (static) data groups, this fact should be obvious. For dynamic data groups, it is sufficient to observe that each mapping can be represented by a regular expression of the form $f.e$ where f is the field name to be traversed and e is the access expression denoting the target group.

Modeling access permissions with data groups is likely to be possible, but tedious. Access permissions are defined structurally without recurse to the class hierarchy, whereas data groups are attached to classes. Hence, access permissions are more flexible to use at the expense of encapsulation. However, the latter could be amended by introducing a naming scheme for abstracting sets of field names in a similar way as data groups. This abstraction could be useful to obtain a more efficient translation of the traversal automata if the names of data groups are globally specified.

6.2 Controlling and Analyzing Effects

There are a number of proposals to control or analyze effects on heap-allocated objects both statically and dynamically.

Some important and strongly-related proposals are based on aspect oriented programming (AOP) or similar techniques. The AspectJ [17] language is the de facto standard for AOP in Java. The pointcut-advice framework of AspectJ is similar to the programming framework of Javassist, but AspectJ operates on the source level. It allows programmers to attach code to method calls and other events during program execution. AspectJ could provide an alternative implementation, but we have not investigated this path.

Beyond AspectJ, DiSL [26], a DSL for bytecode instrumentation poses an alternative to Javassist. DiSL expresses bytecode instrumentations similar to aspects (i.e., independently of the application itself) and has an explicit pointcut/advice model comparable to AspectJ.

Bodden and Havelund [2] propose an algorithm called Racer, an adoption of the Eraser algorithm [28] to Java's memory model, and an AspectJ-based implementation that relies on the three pointcuts `lock()`, `unlock()`, and `maybeShared()` to detect possible data races. While we are not directly targeting race detection, side-effect monitoring can certainly contribute towards detecting unwanted accesses to shared memory.

An effect system is a static analysis that partitions the heap into disjoint regions and annotates the type of a heap reference with the region in which the reference points [11]. Although initially developed for functional languages, region-based effects have been transposed to object-oriented languages [13]. A notable proposal targeting Java is the type and effect system of DPJ [1]. DPJ targets parallel execution and provides a deterministic semantics by default.

A different approach to ensuring disjoint memory accesses relies on unique object references where the typing guarantees that only one reference to a unique object exists. Clarke and Wrigstad [6] provide a good introduction and an overview of several systems.

Compared to previous work on access permissions [16], this implementation relies on a location-based semantics. With this semantics, the installation of a contract requires the evaluation of the path set on the pre-state to a set of locations (objects and field names) tagged with read and/or write permissions, which is linear in the size of the accessible heap in the worst case. However, checking the contract is a simple test for membership in this set. For the path-based semantics in the previous work, installation is a constant-time operation, but checking is linear in the number of installed contracts. Moreover, there are semantics differences, which are touched in Section 2.1 and more thoroughly explained in the cited work.

7. Conclusion

We present a system for fine-grained specification and monitoring of side effects of methods. The specification assigns access permissions to objects based on access paths. In contrast to previous systems, our system enforces a location-based semantics of access paths, where installing a permission requires an object traversal but where checking a permission is a constant-time operation.

To validate our design, we constructed an implementation based on bytecode rewriting at load time or, alternatively, using offline instrumentation. Our evaluation of micro benchmarks demonstrates that the instrumentation of a field access and of an object creation imposes a constant low overhead. It also shows that large visible heaps can lead to a high overhead. Futhermore, the evaluation demonstrates that the overhead imposed by the traversal can be reduced significantly when a contract permits access only to a small number of locations.

Our evaluation of a realistic benchmark suite shows that the overhead for monitoring without installed permissions ranges between about 4% in the `probe` benchmark and 57% in the `illness` benchmark. Installing a non-trivial fully permissive contract which triggers a full traversal on all methods introduces a total overhead of between about 1.66x (`fft`) and 36051x (`boyer`). High overheads occur for contracts covering large parts of a large visible heap.

In future work, we intend to apply our implementations to even larger benchmarks to obtain a better understanding of potential bottlenecks in our implementation. We also aim at benchmarks with more realistic contracts. We further want to improve the performance of the tree traverser by generating specialized traversal algorithms and by using symbolic execution to elide redundant contract installations.

References

[1] R. L. Bocchino Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel Java. In S. Arora and G. T. Leavens, editors, *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 97–116. ACM, 2009. ISBN 978-1-60558-766-0.

[2] E. Bodden and K. Havelund. Racer: Effective race detection using AspectJ. In B. G. Ryder and A. Zeller, editors, *Proceedings of the ACM SIGSOFT International Symposium Software Testing and Analysis*, pages 155–166, Seattle, WA, USA, 2008. ACM. ISBN 978-1-60558-050-0.

[3] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7(3):212–232, 2005. ISSN 1433-2779. .

[4] Y. Cheon. *A Runtime Assertion Checker for the Java Modeling Language*. PhD thesis, Iowa State University, Apr. 2003. TR #03-09.

[5] S. Chiba. Load-time structural reflection in Java. In E. Bertino, editor, *14th European Conference on Object-Oriented Programming*, volume 1850 of *Lecture Notes in Computer Science*, pages 313–336. Springer, 2000. ISBN 3-540-67660-0.

[6] D. Clarke and T. Wrigstad. External uniqueness is unique enough. In L. Cardelli, editor, *17th European Conference on Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 176–200. Springer-Verlag, July 2003. ISBN 3-540-42206-4.

[7] B. Dufour, K. Driesen, L. J. Hendren, and C. Verbrugge. Dynamic metrics for java. In R. Crocker and G. L. S. Jr., editors, *Proceedings of the 2003 ACM SIGPLAN Conference Object Oriented Programming, Systems, Languages, and Applications*, pages 149–168. ACM, 2003. ISBN 1-58113-712-5.

[8] M. Fähndrich, M. Barnett, and F. Logozzo. Embedded contract languages. In S. Y. Shin, S. Ossowski, M. Schumacher, M. J. Palakal, and C.-C. Hung, editors, *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 2103–2110, Sierre, Switzerland, 2010. ACM. ISBN 978-1-60558-639-7.

[9] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In S. Peyton-Jones, editor, *Proceedings International Conference on Functional Programming 2002*, pages 48–59, Pittsburgh, PA, USA, Oct. 2002. ACM Press, New York.

[10] R. Floyd. Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics*, pages 19–32, 1967.

[11] D. Gifford and J. Lucassen. Integrating functional and imperative programming. In *Proceedings of the 1986 ACM Conf. on Lisp and Functional Programming*, pages 28–38, Cambridge, Massachusetts, United States, 1986. ACM Press.

[12] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. *The Java Language Specification, Java SE 7 Edition*. Addison-Wesley, 2013. ISBN 0133260224.

[13] A. Greenhouse and J. Boyland. An object-oriented effects system. In R. Guerraoui, editor, *13th European Conference on Object-Oriented Programming*, volume 1628 of *Lecture Notes in Computer Science*, pages 205–229, Lisbon, Portugal, June 1999. Springer-Verlag. ISBN 3-540-66156-5.

[14] P. Heidegger and P. Thiemann. A heuristic approach for computing effects. In J. Bishop and A. Vallecillo, editors, *Proceedings of the 49th International Conference on Objects, Models, Components and Patterns*, volume 6705 of *Lecture Notes in Computer Science*, pages 147–162, Zurich, Switzerland, June 2011. Springer. ISBN 978-3-642-21951-1.

[15] P. Heidegger and P. Thiemann. JSConTest: Contract-driven testing and path effect inference for JavaScript. *Journal of Object Technology*, 11 (1):6:1–29, Apr. 2012. .

[16] P. Heidegger, A. Bieniusa, and P. Thiemann. Access permission contracts for scripting languages. In J. Field and M. Hicks, editors, *Proceedings 39th Annual ACM Symposium on Principles of Programming Languages*, pages 111–122, Philadelphia, USA, Jan. 2012. ACM Press.

[17] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In J. L. Knudsen, editor, *15th European Conference on Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag. ISBN 3-540-42206-4.

[18] R. Kramer. iContract — the Java design by contract tool. In *Proceedings of the Technology of Object-Oriented Languages and Systems*, pages 295–307, Santa Barbara, CA, USA, 1998. IEEE Computer Society.

[19] G. Leavens et al. Jml reference manual. `http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman.html`, May 2013.

[20] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188, Norwell, MA, USA, 1999. Kluwer Academic Publishers.

[21] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1-3):185–208, 2005.

[22] H. Lehner. *A Formal Definition of JML in Coq and its Application to Runtime Assertion Checking*. PhD thesis, ETH Zurich, Switzerland, 2011.

[23] H. Lehner and P. Müller. Efficient runtime assertion checking of assignable clauses with datagroups. In D. S. Rosenblum and G. Taentzer, editors, *Proceedings of the Fundamental Approaches to Software Engineering: 13th International Conference*, volume 6013 of *Lecture Notes in Computer Science*, pages 338–352, Paphos, Cyprus, 2010. Springer. ISBN 978-3-642-12028-2.

[24] K. R. M. Leino. Data groups: Specifying the modification of extended state. In *Proceedings of the 13th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 144–153, Vancouver, BC, Canada, Oct. 1998. ACM Press, New York. ISBN 1-58113-005-8.

[25] K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *PLDI*, pages 246–257, Berlin, Germany, June 2002. ACM Press. ISBN 1-58113-463-0.

[26] L. Marek, A. Villazón, Y. Zheng, D. Ansaloni, W. Binder, and Z. Qi. DiSL: a domain-specific language for bytecode instrumentation. In R. Hirschfeld, É. Tanter, K. J. Sullivan, and R. P. Gabriel, editors, *Proceedings of the 11th International Conference on Aspect-Oriented Software Development*, pages 239–250, Potsdam, Germany, 2012. ACM. ISBN 978-1-4503-1092-5.

[27] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.

[28] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.