Runtime Verification of Object Lifetime Specifications

by

Zev Benjamin

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Computer Science and Engineering

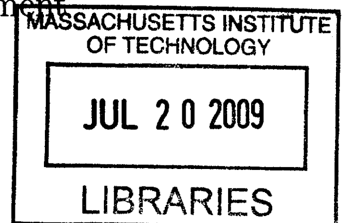
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2009

© Zev Benjamin, MMIX. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.



Author . . . .
Department of Electrical Engineering and Computer Science
May 22, 2009

Certified by . . . .
Daniel Jackson
Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

ARCHIVES

Runtime Verification of Object Lifetime Specifications

by

Zev Benjamin

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2009, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science and Engineering

Abstract

This thesis reports on the implementation of a runtime verification system for *object lifetime specifications*. This system is used to explore and evaluate the expressiveness object lifetime specifications.

Object lifetime specifications are program annotations that indicate, in terms of program execution, when objects should be reclaimed. Object lifetime specifications are designed to be used within the context of an object-oriented programming language with automatic storage reclamation (*i.e.*, garbage collection). Our runtime verification system observes program execution and reports objects that are not reclaimed before the end of their specified lifetime.

We implement our system with the *Java Path Finder* model-checking framework. JPF supports modeling non-determinism in programs. We leverage this support to verify all possible thread interleavings when checking user programs.

Thesis Supervisor: Daniel Jackson
Title: Professor

Acknowledgments

I dedicate this work to my parents. They have taught me so much and have been so supportive.

I thank my advisor Daniel Jackson for his invaluable insight and suggestions. Finally, this thesis would not have been possible without Derek Rayside, a Ph.D. candidate in the Software Design Group, who invented Object Lifetime Specifications, collaborated with me on the implementation of JunkAware, and who guided me in the writing of this thesis.

Contents

1 **Introduction** 6

1.1 Related Work 7

1.2 JunkAware 9

2 **Motivating Examples (Anti-Patterns)** 10

2.1 Lapsed Listeners 10

2.2 Crufty Caches 11

2.3 Sneaky Strings 11

2.4 Accumulating Arrays 12

2.5 Inheritance Irritation 14

2.6 Tangled Ownership Contexts 14

2.7 Bloated Facade 14

2.8 Construction of Zombie References 15

3 **Object Lifetime Specifications** 16

3.1 Groups 17

3.2 Annotations 18

3.2.1 Group Creation 18

3.2.2 Bounding Group End-time 19

3.2.3 Pinning 20

3.3 Group and Object Lifetimes 21

3.4 Usage Methodology 22
3.5 Notes 23

4 Implementation of a Runtime Verifier for Object Lifetime Specifications using Java Path Finder 25

4.1 Java Path Finder 25
4.2 Overview 26
4.3 Verifier State 28
4.4 IDMap 29
4.5 Reading the Specifications 30
4.6 Classloader Activity 31
4.7 Implicit Groups 31
4.8 Backtracking 32
4.9 Managing Specification Verifier State 33
4.10 Limitations 35

5 Implementation Alternatives 37

5.1 JDI 37
5.2 Jikes 38
5.3 Production VM 38

6 Evaluation 40

7 Conclusion 46

Chapter 1

Introduction

Memory leaks are a common class of errors in computer software and often lead to bugs or other undesirable program behavior. Traditionally, memory leaks are considered in the context of languages with explicit memory management, such as C. In such contexts, a memory leak is usually defined as a case where an object is allocated, but can never be deallocated because all references to the object have been destroyed. This situation leaves the object persistent in memory until program termination. Such objects are called *garbage*, and there are well-known tools for detecting when this sort of problem arises [3, 7, 10].

However, memory leaks also occur in garbage collected languages such as Java. Like in non-garbage collected languages, the memory leaks we focus on in Java are semantic errors on the part of the programmer (rather than bugs in the JVM). In a garbage collected language, memory leaks occur not because references to unneeded objects are lost but rather because references to unneeded objects are retained. We call these objects *junk*.

The goal of this project was to build a tool that allows programmers to use Object Lifetime Specifications included in their Java code via annotations to dynamically detect the presence of junk objects created during runtime.

1.1 Related Work

There are two existing classes of solutions to the problems of junk. The first attempts to prevent the retention of junk, and the second attempts to detect the presence of junk after the program has been written.

Most of the solutions in the preventative category take the form of stack disciplines. For example, one can disallow dynamic memory allocation entirely, which guarantees that junk objects are not created because all objects in a stack frame are destroyed upon that frame's exit. The Real-Time Specification for Java (RTSJ) [1] is slightly less restrictive: it allows dynamic memory allocation, but disallows objects from referencing objects at a deeper stack frame. This scheme also guarantees that objects from a stack frame can be deallocated upon that frame's exit.

The problem with these methods is that they are too restrictive. Disallowing dynamic memory and the solution used by RTSJ both greatly limit expressibility. Worse, these disciplines are almost always incompatible with libraries, forcing the programmer to use non-standard utility code. While possibly acceptable for mission-critical code, these are not good general-purpose solutions.

In the detection category, there are several strategies used by different tools. Röjemo and Runciman [8] used staleness (the time between when a object is last used and when it becomes garbage) as an approximation for junk; Leakbot [6] uses heap snapshots to infer relationships between objects; and Cork [5] examines the heap to infer ownership between types.

Other research, inspired by linear types [14], seeks to restrict the users of an object/group so that releasing it is no longer potentially global. The trade-off of such approaches is that they restrict sharing. Our approach does not restrict sharing, but requires the programmer to characterize the sharing when its lifetime does not conform to a relaxed stack discipline.

For the programmer, object lifetime specifications are intuitively similar to re-

gions, but with a few important differences that increase their expressiveness. Object lifetime specifications are written in terms of *groups* of objects, which are defined by the programmer. Groups differ from the conventional notion of regions in the following ways:

- Groups may be shared across threads.
- The expected lifetime of a group can be specified in flexible ways. Most memory-safe approaches to regions follow a stack discipline, where the region expires with the method frame that created it (although there are exceptions to this rule, *e.g.*, [4]). This limits most implementations of regions (including [4]) to single-threaded programs.
- An object may be a part of multiple groups. With all approaches to regions that we are aware of, an object may be a part of exactly one region.
- Groups are an analysis time concept: they are not used to actually release objects at runtime. The regular garbage collector still manages memory at runtime. By contrast, regions are usually used to physically manage memory.

These existing tools either provide strong guarantees, but limit expressiveness and carry a large annotation burden, or provide weak guarantees by using heuristics, and hence require no annotations. For example, the staleness heuristic, while a good indicator of junk, can both over-estimate and under-estimate the junk in the program. Objects that are not used in a particular run of a program (perhaps because they are needed for functionality that the user did not ask for) might erroneously be identified as junk. Alternatively, junk objects in a hash table serving as a cache might not be identified because the hash table will rehash its elements when its size gets too large. If an array was used for the cache, the elements would have been identified as junk when they became stale. However, when the hash table rehashes, it will call the `hashCode()` method on all of its elements, freshening them.

1.2 JunkAware

JunkAware is a runtime verifier for Object Lifetime Specifications. JunkAware and Object Lifetime Specifications together attempt to target a middle ground. As discussed above, most preventative solutions are overly restrictive, and most heuristic solutions don't give any guarantees. We trade very mild restrictions on expressiveness for guarantees on single executions (as opposed to a static guarantee on all possible executions).

Further, our tool is completely noninvasive to normal program execution. User programs are marked up with Java *annotations*, special program text that can be used to add extra information to classes, methods, and fields. These annotations will be ignored except when the program is run under JunkAware.

Our tool can also be applied incrementally to programs. Users don't need to add any annotations before running it under the tool. Then, annotations can be added at a very coarse granularity first. If required, users can search for leaks at finer and finer granularities by adding more annotations.

Chapter 2

Motivating Examples (Anti-Patterns)

In this chapter, we exhibit several anti-patterns that are common causes of memory leaks.

2.1 Lapsed Listeners

The Observer pattern [2] is often used in Java and is especially prevalent in AWT and JavaBeans. It is also central in the MVC [2] paradigm. However, it can also be a subtle source of memory leaks [12, §6][9, §2.2]. Observers are called Listeners in Bitter Java.

The point of the pattern is to separate the object that updates a value from the object that uses the value. One or more *observer* objects register with the *subject* to receive updates. The problem comes when the observers fail to unregister themselves. Even if all other references to the observer are explicitly destroyed, the subject still holds on to a reference. Worse, the user of the observer might not even know which subjects it is observing to unregister it.

The author of Bitter Java also notes that often times the unregister step is omited

in code snippets for brevity. Thus, programmers who copy and paste the code forget to unregister their observers.

2.2 Crufty Caches

In a crufty cache, a memoized call becomes the cause of a memory leak. For example, an interning factory could be a crufty cache. The problem is that the cache has no way of knowing when to remove the items from its cache.

While the diagnosis for crufty caches is similar to lapsed listeners and other anti-patterns, the fixes are different. For crufty caches, one solution is for the cache to use weak or soft references. Thus, if no other object references the cached item, it will become a candidate for garbage collection. However, this solution does not work for lapsed listeners and others, as the references are often required because they cannot be recomputed as can the objects stored in a cache.

2.3 Sneaky Strings

In many string-processing applications, the goal is to extract certain strings or to rearrange text. It is common to read in the entire input and pull out the pieces desired using the substring() method. The way substring() is implemented, the returned String object is actually just a pointer to the original string with an offset into it. This method of creating substrings is efficient, but the memory for the original string is preserved.

In an application that pulls out the first word of each line of the file, the entire file will persist in memory even though the text of the other portions are inaccessible. An example of such a program is shown in Figure 2-1. The leak occurs on line 14. The solution to this problem is to use the String constructor with another string as its argument. Adding the following after line 14 would eliminate the leak:

```
substring = new String(substring);
```

Figure 2-1: An example of sneaky strings.

```
1 public class SneakyStrings {
2   public static void main(String[] args) throws IOException {
3     String fileName = "analysands/antipatterns/foo.txt";
4     List<String> firstWords = firstWords(fileName);
5     System.out.println(firstWords);
6   }
7
8   static List<String> firstWords(String fileName) throws IOException {
9     List<String> results = new LinkedList<String>();
10    BufferedReader r = new BufferedReader(new FileReader(fileName));
11    String line;
12    while ( (line=readLine(r)) != null) {
13      int index = line.indexOf(' ');
14      String substring = line.substring(0, index);
15      results.add(substring);
16    }
17    return results;
18  }
19
20  static String readLine(final BufferedReader r) throws IOException {
21    return r.readLine();
22  }
23 }
```

2.4 Accumulating Arrays

Consider an ArrayList implementation that does not set appropriate array indices to null when an element is removed from the list. This might be done for efficiency reasons.

The leak actually takes place on line 22. After pop() is invoked, the bounds checking code in get() ensures that the popped item is inaccessible. However, the appropriate entry in the underlying array is not set to null, which prevents the object from being a candidate for garbage collection.

Figure 2-2: An example of a leaky ArrayList implementation.

```java
public class MyArrayList<E> {
  E[] array = new E[0];
  int size = 0;

  public void add(E elem) {
    E[] tmp = new E[size + 1];
    System.arraycopy(array, 0, tmp, 0, size);
    tmp[size] = elem;
    array = tmp;
    size++;
  }

  public E get(int index) {
    if (index >= size || index < 0) {
      throw new ArrayIndexOutOfBoundsException(index);
    }
    return array[index];
  }

  public void pop() {
    if (size > 0) {
      size--;
    }
  }
}
```

2.5 Inheritance Irritation

Inheritance irritation [9, §2.1] arises from improper use of inheritance. Suppose there is some superclass A that has a private field. B inherits from A but doesn't use the functionality of the private field. It should set the field to null in order to free the memory associated with it. However, because it is private, B cannot access the field. The solution is to extend an appropriate interface, instead.

2.6 Tangled Ownership Contexts

The idea of tangled ownership contexts mostly makes sense in the context of object ownership graphs and was described by Rayside and Mendel [9], §2.4. However, they are still causes of memory leaks.

The problem arises when objects are not owned by any one object, but are shared. For example, suppose the program creates two parse tree objects. If, due to some bug, node objects are actually shared between these parse trees (*i.e.*, the same node or subtree appears in both trees), then keeping a reference to one parse tree may keep part or all of the other from being released.

2.7 Bloated Facade

In the bloated facade [9, §2.5], an object keeps a pointer to an object A because it needs to access data from it. The problem is that A also holds pointers to other pieces of memory. In the bloated facade, the data that is required is not encapsulated into its own object, so the user cannot just keep a pointer to that. The solution to this problem is to factor out the required data into its own object.

2.8 Construction of Zombie References

Rayside and Mendel [9] note that once junk objects have been created, it is possible to exacerbate the problem. For example, if object A points to the junk object B, object C might also create references to B solely based on the fact that A has a reference to it. This will ultimately just make B harder to get rid of.

Chapter 3

Object Lifetime Specifications

The idea of Object Lifetime Specifications is not a contribution of this thesis, but this exposition of them is. The material in this chapter is based on personal communication with Derek Rayside.

We think of a Java *program* as made up of *threads*, *stack frames*, and *objects*. We will refer to each of these elements as a *shard,* and to the program, thread, and stack frame shards collectively as *computational shards.* The *end-time* of a shard is easily observable:

- An object's end-time is the point at which there are no more references to it.
- A stack frame's end-time is either when its associated return statement is reached or an exception is thrown an not caught by the frame.
- A thread's end-time is when its initial stack frame ends.
- A program's end-time is when its final thread ends.

We call a shard whose end-time is exactly when the programmer desires *timely.*

Unfortunately, a shard's end-time may occur later than the programmer desires. We call these shards *tardy* (thus, "junk" is another name for a tardy object). To

resolve such discrepancies, the programmer must first observe at a macro level that some shard is tardy. Then he must observe at a micro level which instance of the shard is tardy. Finally, he must reason about the tardy shards in order to find a strategy for making them timely.

These observations are relatively easy for computational shards because they usually impact external events. Once these observations have been made, the strict hierarchy among computational shards serve as a powerful reasoning framework.

However, the observations for objects are more difficult. When a computational shard is tardy, it is the currently active computational shard. In contrast, a tardy object is one that is *not* involved in the current computation. Hence, the observations are hard.

Further, objects can be shared, and thus do not naturally form a hierarchy. Therefore, even if the programmer identifies an object as tardy, he lacks a reasoning framework to strategize about making it timely.

Object Lifetime Specifications tie expected object end-times to stack frames, thereby extending the powers of observation and reasoning enjoyed by computational shards to objects.

3.1 Groups

We tie expected object end-times to stack frames by associating every object with a *group*. A group is simply a collection of objects in the running program. An object is associated with a group at the time it is allocated.

Every group has a name that consists of a *namestem* and an *index*. The namestem is a programmer-provided string, and the index is an integer assigned by the system. The programmer associates a namestem with a method in their Java program. Each time that method is invoked, a new group is created with the associated namestem

and the next index. Thus, for each thread there is a stack of groups that parallels its call stack. When a object is allocated by a thread, it is placed in the group at the top of its stack.

Object Lifetime Specifications operate on the level of groups. Operating at this coarse granularity reduces the annotation burden on the programmer.

3.2 Annotations

We specify Object Lifetime Specifications for Java programs using Java annotations. The annotations we use are listed in Figure 3-5.

3.2.1 Group Creation

There are two kinds of creation annotations, each of which declare that a new group shall be created when the annotated method is called. They are called @Creates and @CreatesTemp. @Creates takes a namestem as an argument, while @CreatesTemp takes no arguments.

Imagine a compiler that takes some program text as input, possibly optimizes it, and produces native code as output. A snippet of such a compiler which demonstrates the @Creates annotation is shown in Figure 3-1. Here, the buildAST() method has an annotation with the "astVars" namestem as its argument. This means that when buildAST() is called, a new group named "astVars_i" will be created and pushed on to the group stack, where i is the number of times a group with the "astVars" namestem has been created before. Thus, the ast variable will belong to the "astVars_i" group. The variable text will not be assigned to the new group because the object it points to was created outside of the method call.

The @CreatesTemp annotation is equivalent to declaring both a @Creates and @ReleasesAll (see below) annotation with the same namestem.

Figure 3-1: Example usage of the @Creates annotation.

```
1 @Creates("astVars")
2 public void buildAST(String text) {
3   ast = new Tree();
4   ...
5 }
```

3.2.2 Bounding Group End-time

There are two kinds of annotations for bounding the end-time of groups: @Releases and @Retains. Each of these annotations has several variations that identify exactly which groups are affected. All variations take a namestem as an argument.

The variations for @Releases are Oldest, Newest, Some, and All (*i.e.*, the annotations are @ReleasesNewest, @ReleasesOldest, *etc.*). The variations for @Retains are Oldest, Newest, and All. The Oldest variation indicates that the annotation refers to the group with given namestem and the lowest index; the Newest variation to the one with the given namestem and the highest index; the Some variation to any group with the given namestem; and the All variation to all groups with the given namestem.

If the programmer knows exactly when the program is done with some group, they can specify that with a @Releases annotation. However, if the programmer is not sure exactly what other parts of the program may be using some group at a given time (*e.g.*, due to concurrency), then they can simply mark the parts of the program that are expected to extend the lifetime of the group with @Retains, and our dynamic analysis will observe which of these @Retains methods ends last.

Continuing our compiler example, the code generation method, which demonstrates a @Releases annotation, is shown in Figure 3-2. The emitCode() method has a @ReleasesAll annotation with the "astVars" namestem. This indicates that all objects in all groups with the "astVars" namestem should be deallocated after emitCode() has finished. This is done in the example by setting the ast variable to null.

Finally, the optimization step of our compiler, which makes use of a @Retains

Figure 3-2: Example usage of a @Releases annotation.

```
1 @ReleasesAll("astVars")
2 public void emitCode() {
3   walkTree(ast);
4   ast = null;
5   ...
6 }
```

annotation, is shown in Figure 3-3. The optimizeAST() method has a @RetainsAll annotation with the "astVars" namestem. This indicates that if optimizeAST is called, all objects in groups with the "astVars" may persist at least until the end of the method.

Figure 3-3: Example usage of a @Retains annotation.

```
1 @RetainsAll("astVars")
2 public void optimizeAST() {
3   constantProp();
4   elimDeadCode();
5   ...
6 }
```

3.2.3 Pinning

Finally, there is a @Pins annotation for specifying a simple form of object ownership. @Pins is placed on a field instead of a method. It indicates that in any instance of an object, that field should be a member of all groups that that object is a member of.

For example, in Figure 3-4, the data field is pinned to instances of the Cell class. This is because the data is created elsewhere, and thus can belong to a different group than the Cell. The @Pins annotation ensures that the data's expected lifetime is at least as long as its corresponding Cell.

Figure 3-4: Example usage of the @Pins annotation.

```
1 public class Cell {
2   @Pins
3   private Object data;
4
5   public Object getData() {
6     return data;
7   }
8   public void setData(Object obj) {
9     data = obj;
10  }
11 }
```

3.3 Group and Object Lifetimes

The end-time of a group is either specified by an @Releases annotation that names the group, or the maximum expected end-time of the set of @Retains annotations that name the group.

The expected end-time of an object is the maximum expected end-time of all of the groups it is a member of. Objects may be members of multiple groups only due to pinning.

Recall our example compiler from Section 3.2. While we don't have enough of the compiler to determine exactly how it behaves, we can reason about how it should behave based on its annotations. If the user requests a full compilation of a program, then the ast should be released after the emitCode() method. This is because the @ReleasesAll annotation on emitCode() requires all objects in an "astVars" group to be released after that method call.

However, suppose the user only requests a parse and optimization, without code generation. In this case, the ast should be released after the optimizeAST() method because optimizeAST() is the last method called that @Retains an "astVars" group.

Finally, if the user only requests a program parse (perhaps to just check syntax), the ast should be released after the buildAST() method because no other method is called that @Retains or @Releases an "astVars" group.

Figure 3-5: Annotations to specify object lifetimes

Annotation	Entity	Description
@Creates("group")	method	Objects allocated by this method's call tree are members of the named group.
@CreatesTemp	method	Objects allocated by this method's call tree are temporary, and should be released by the time this method finishes.
@Releases{Oldest, Newest, Some, All}("group")	method	Objects in the identified group should be released by the time this method finishes.
@Retains{Oldest, Newest, All}("group")	method	The permissible lifetime of objects in the identified group extends at least as long as the end of this method's execution.
@Pins	field	Superobject pins subobject: subobject becomes a member of all groups that the superobject is a member of.

For any of these three possible inputs, if the program fails to behave as annotated, JunkAware will report that the ast object has become junk.

3.4 Usage Methodology

We imagine an incremental usage methodology for object lifetime specifications, in the following sense: the programmer may start with a program with no annotations, and quickly develop a set of annotations that characterize the program accurately but coarsely. Then, if desired, the programmer can add more annotations to increase precision. These annotations will generally be added in a top-down manner, by starting at the program entry point(s).

The object lifetime specification for a program with no annotations is that all objects should be releasable by the time the thread that allocated them has completed.

For single threaded programs this extremely coarse specification holds trivially. However, this specification will not hold for many concurrent programs. In particular, for GUI programs, the main thread usually constructs the GUI and then terminates, leaving the GUI framework's event processing thread holding the objects. For such programs, a single annotation – a @Retains or @Releases of the system thread group on the GUI's dispose() method – will suffice to indicate that the objects constructed by the main thread are shared with the GUI event processing thread.

The practical utility of these extremely coarse default specification is, of course, limited. However, they have methodological value: the programmer starts from a position where they have an accurate specification, and simply work on making it more precise, as their needs, available time, and level of interest dictate. This approach correlates results with programmer effort: the more work the programmer invests, the more precise a result they get — but they always have a specification that's consistent with their program.

3.5 Notes

An object lifetime specification is not an object access specification. It is important to note that an object *lifetime* specification is not an object *access* specification: a lifetime specification indicates a bound on the object's lifetime; an access specification indicate a bound on what can access the object. Consider the case of the observer design pattern: the subject needs to have access to all of the observers, in order to notify them of updates; however, the lifetime of the observers is, in most cases, shorter than the lifetime of the subject.

Non-termination. JunkAware deals with *partial correctness*: *i.e.*, if the program terminates, then the judgments are accurate. Non-termination, whether due to deadlock, looping, continuation-passing style, or something else, may prevent JunkAware

from making a judgment about the presence of leaks in the program.

Chapter 4

Implementation of a Runtime Verifier for Object Lifetime Specifications using Java Path Finder

The runtime verifier JunkAware is implemented on top of the Java Path Finder (JPF) virtual machine. This allows us to monitor analysand program events such as object creation and destruction. In turn, this allows us to check the specifications as the program is running.

4.1 Java Path Finder

Java Path Finder [13] is a special Java virtual machine that is primarily designed for model checking, but can also be used for a variety of other similar kinds of analyses.

In JPF, in contrast with other JVM's that are written primarily for performance rather than analysis, it is possible to force the garbage collector to run. We exploit

this to observe object deallocation by forcing a garbage collection after each method call. Thus, any objects that can be deallocated at that point, will be.

JPF can handle non-determinism in programs by exploring all the possible alternatives. It does this efficiently by exploring the program state space using backtracking. We discuss how we use this feature in Section 4.8.

For our purposes, JPF has two important concepts: *listeners* and *properties*. Listeners inform a piece of analysis software, such as JunkAware, when important events have occurred in the analysand program. JPF has two kinds of listeners: VMListeners and SearchListeners. VMListeners are informed of virtual machine activity, such as object creation or instruction execution. SearchListeners are informed of activity concerning the exploration of the program execution state space. For example, SearchListeners are informed when JPF backtracks or when the program is finished executing.

Properties are objects that represent invariants that the analysis program would like to check. For example, JPF comes with a property for checking the invariant that no uncaught exceptions are thrown. JunkAware checks the invariant that the program has no junk objects. JPF periodically invokes the property's check method and, if it returns false, reports the property failure to the user.

4.2 Overview

Although backtracking is an important feature of the JPF implementation, it significantly complicates the design of JunkAware. Therefore, we will first describe a simplified design in the absence of backtracking. An design rationale of the simplified architecture is shown in Figure 4-1, and the major components of the tool are summarized in Table 4.1.

JunkAware interfaces with JPF at two points: it registers the singleton JunkLis-

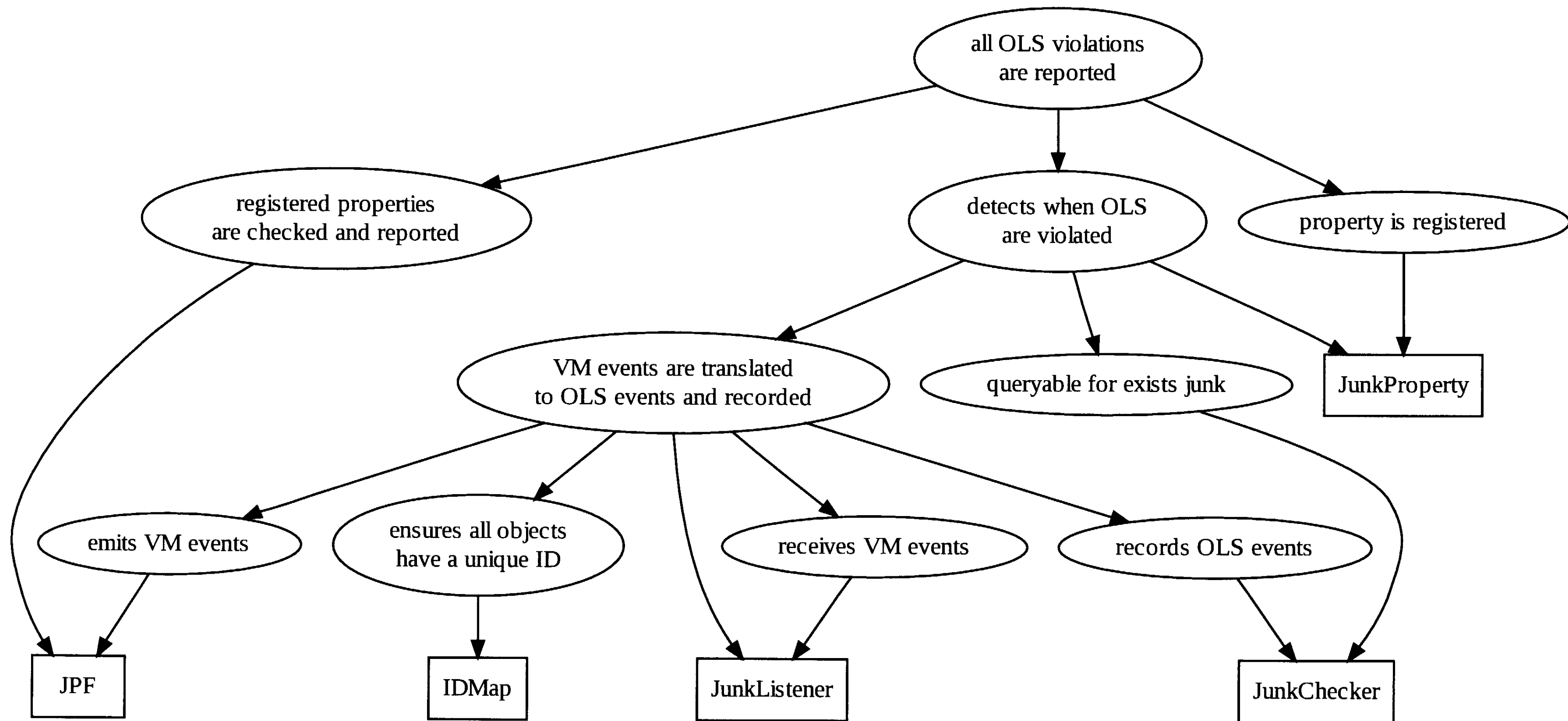


Figure 4-1: Design rationale [?] diagram without backtracking. Boxes represent system components and ellipses represent properties. Arrows show which components or properties are required to satisfy others.

Component	Description	Section
JunkListener	an instance of VMListener and SearchListener that receives events about the current program execution from JPF	
JunkProperty	an instance of JPF's GenericProperty that we use so that we can report memory leak events with JPF's generic reporting facility.	
VerifierState	encapsulates the current state of the verifier.	4.3
IDMap	a utility class that we use to ensure objects have unique IDs.	4.4
JunkChecker	enables JunkAware to deal with JPF's backtracking.	4.8

Table 4.1: The major components of JunkAware. Each is discussed more fully in the indicated section.

tener with JPF as a VMListener and it registers JunkProperty with JPF as a property to check. When JunkAware is run, it registers these objects and then instructs JPF to begin running the target program. As JPF executes the program, it will notify JunkListener of VM events. The JunkListener will then use these events to manage the verifier state via the VerifierState. JPF will also periodically query the JunkProperty to determine whether it has been violated (*i.e.*, whether there are junk objects). JunkProperty can answer this query by forwarding it to the VerifierState.

The IDMap is used by the VerifierState to ensure that all objects referenced by the verifier state are identified by a unique ID (see Section 4.4).

4.3 Verifier State

The main datastructure in our system is the VerifierState object. It encapsulates the current state of all objects in the running program. It has the following eight operations:

- objectCreated(threadID, objectID)
- objectReleased(objectID)
- pushGroup(threadID, namestem)

- popGroup(threadID, namestem)
- pin(threadID, superObjectID, subObjectID)
- associateObjectWithGroup(objectId, namestem)
- pushAuthority(threadID, authority)
- popAuthority(threadID, authority)

The first two operations record object creation and release. The pushGroup and popGroup operations create and close a group, respectively; the verbs in the names reflect the fact that groups follow the call stack. The VerifierState is responsible for incrementing the index associated with the given namestem and concatenating the two to name the new group. The pin operation indicates that some superobject pins some subobject. The associateObjectWithGroup operation allows the tool to arbitrarily assign objects to a group. This is mainly used for system objects. The last two operations are used for the @Releases and @Retains annotations, which are abstracted to the notion of an *authority* (*i.e.*, an authority to extend the lifetime of some group of objects).

4.4 IDMap

For our analysis, we require two properties of the way JPF identifies objects. The ID of an object must not change over its lifetime. Additionally, no two objects may simultaneously have the same ID. JPF, as with most virtual machines, provides these properties. However, also as with most other virtual machines, JPF does not guarantee that object IDs will not be reused for future objects. Our analysis requires unique IDs across the entire program execution.

Thus, we must transform the object IDs that JPF provides into IDs that are unique across the program execution. The IDMap module provides this service. The IDMap has the following operations:

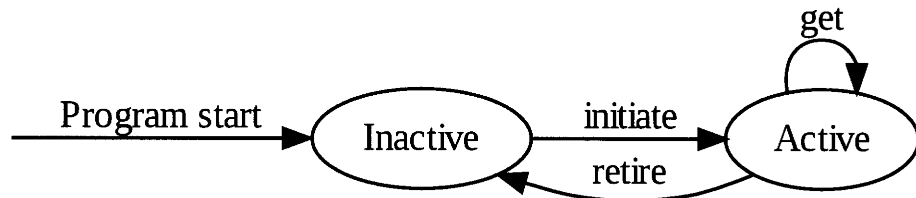


Figure 4-2: State machine representation of the life of an ID in the IDMap. Nodes are ID states and edges are operations with that ID as an argument. Transitions not present are illegal.

- initiate(jpf-oid)
- retire(jpf-oid)
- get(jpf-oid)

The new identifier that the IDMap creates is the concatenation of the JPF object ID with a *generation number*.

Each JPF ID is associated with an instance of the state machine shown in Figure 4-2. When a new JPF object ID is encountered (usually during object creation), initiate is called with that ID. This increments the generation number for that ID and makes subsequent calls to get with that ID valid. After an ID has been initiated, the get operation will map that ID to a unique identifier. When an object is destroyed, retire is called with the destroyed object's ID. This makes subsequent calls to get invalid.

4.5 Reading the Specifications

We read the specifications from the annotations in the source files by registering the VM listener AnnotationExtractor. This listener only listens for the classLoaded event.

When it receives such an event, it walks over the newly loaded class' methods and reads the annotations. The annotations found are recorded for later verification.

In the case of JDK library code, we cannot actually alter the code to add our annotations. Therefore, we include a separate text file that indicates `@Pins` annotations that would have been declared in several JDK classes.

4.6 Classloader Activity

Like many other virtual machines, JPF loads classes on demand. When it encounters an instantiation or static method invocation of a class it has not yet seen, it automatically invokes the classloader to load the class. JPF does this by forcefully pushing the frame for the classloader onto the stack and executing it in the current user thread. This would cause (potentially long-lived) objects created during the classloading process to belong to whatever group was at the top of the group stack at the time the classloader was invoked. Because the programmer can't anticipate when the classloader will be invoked, he cannot prevent these objects from becoming members of arbitrary groups and later reported as junk without the assistance of JunkAware.

To solve this problem, we provide JPF with a custom StaticArea (the class JPF uses to represent part of the heap). The StaticArea is informed when new classes are loaded via its addClass method. Our special StaticArea brackets calls to this method with calls to temporarily disable Object Lifetime Specifications checking in the thread that is loading the new class.

4.7 Implicit Groups

In addition to user-named groups, there are two kinds of groups that are implicitly defined: groups for threads and the distinguished system group. A group is always

pushed onto a thread's group stack just before the thread starts running. The thread object itself is a member of this group to prevent it from being reported as junk. These groups for threads behave like user-named groups in all other respects. The system group is a special catch-all group. Objects allocated in native code or otherwise outside normal creation methods belong to the system group. The end-time of the system group is the end-time of the program.

There is also a special group for the main thread. It is pushed on to the main thread's group stack before the program starts.

4.8 Backtracking

JPF supports *backtracking* to explore all possible program states for a given program input. Every time JPF encounters a non-deterministic choice in program execution, it stores the program state so it can return to that point of execution. It then proceeds to explore all possible choices at that point.

One source of non-determinism that we are particularly interested in is that of thread-scheduling. We conjecture that one of the reasons that some memory leaks seem to only manifest once the program has been deployed is because they only occur on some uncommon thread schedules. We similarly expect that there will be cases where a programmer writes an object lifetime specification that the program conforms to when the program executes its most common thread interleaving, but that the program will have some less common interleavings where it does not conform to the programmer's expected object lifetime specification. JPF helps us explore these empirical concerns.

In order to make this exploration not prohibitively expensive, JPF collapses the state space. That is, if two branches lead to the same state, JPF will only explore the possibilities at that new state once.

To support JPF's backtracking, we add a layer of indirection between the JunkListener and the VerifierState called the JunkChecker. The design rationale for the version of JunkAware that supports backtracking is shown in Figure 4-3.

The JunkChecker stores a stack of ⟨VerifierState, IDMap ⟩ pairs (we instruct JPF to explore the state space depth-first). Because the JunkListener receives notifications of when the search has branched or backtracked, it can instruct the JunkChecker to push or pop these pairs.

4.9 Managing Specification Verifier State

One complication with supporting JPF's backtracking is that it is based on the running program's memory state. However, relying solely on the running program's state is not sufficient for our purposes. We need to include the state of the verifier. To illustrate this, consider the following example program.

After some call either to A() or B(), the heap state is the same. In both cases, there is an object of type Nugget in the heap with a value of 1. However, if A() was called, this object belongs to the "foo" group. If B() was called, the object instead belongs to some other group (possibly a thread's group). It would be undesirable for JPF to treat these situations as the same for the purposes of backtracking.

We considered two alternative ways to overcome this. We could have stored the verifier state inside the heap of the running program. This would have required no modification of the JPF state space code, but would have required instrumenting user programs.

Instead, we decided to provide JPF with our own heap serializer that was aware of the verifier state. Thus, at any given point in the program, when JPF wishes to serialize the heap, we also append a serialized version of the current MemspecState object.

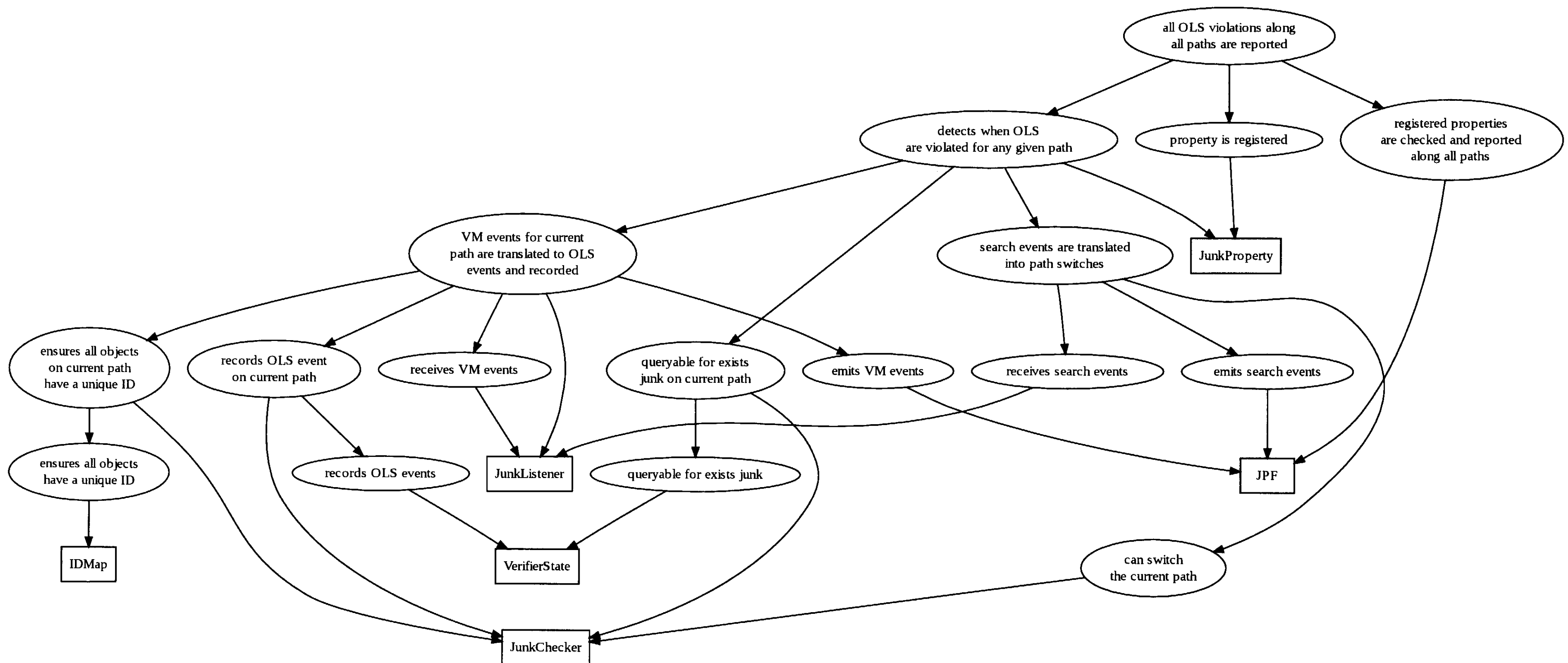


Figure 4-3: Design rationale diagram with backtracking. Boxes represent system components and ellipses represent properties. Arrows show which components or properties are required to satisfy others.

One problem with this approach is that we make no attempt to canonicalize the verifier state. Thus, it's possible for two identical program states to appear as different states to JPF. While this makes the state space exploration less efficient, it is still correct.

4.10 Limitations

Our tool has some limitations imposed on it because it uses JPF, and one minor engineering limitation. While JPF does support a large portion of the JDK libraries, there is also a significant portion missing. We believe this is at least partially because implementing versions of these libraries that do backtracking correctly is difficult. Thus, the number of real-world programs we can run under our tool is currently somewhat limited.

Our tool also does not currently handle exceptions correctly. We do not believe that this is a major issue, however, as exceptions should be rare in correctly executing programs.

Figure 4-4: Two functions that leave the heap in an identical state.

```
1 public class Foo {
2   public class Nugget {
3     final int value;
4     Nugget(final int value) { this.value = value; }
5   }
6   public Nugget heapData;
7
8   @CreatesGroup{"foo"}
9   public void A() {
10    heapData = new Nugget(1);
11  }
12
13  public void B() {
14    heapData = new Nugget(1);
15  }
16 }
```

Chapter 5

Implementation Alternatives

We ultimately chose Java Path Finder as our monitoring VM, but there are several other technologies which we could have used. We explain some of those alternatives below.

5.1 JDI

The Java Debugger Interface is an API for interacting with the Java debuger from another Java process. Here, we would have used the regular Java VM, except we would have controlled execution using the debugger.

Using JDI would have had many advantages. First, since most Java programs target the Sun JDK, our tool would have had no problem running third-party code. JDI allows the full expressive power of Java.

However, the JDI does have some drawbacks. Because it is primarily used to debug Java programs and not to study program execution, it does not expose all of the primitives we would like. For example, it does not allow observation of array reads and writes. Using the JDI would probably have also required us to instrument the running program. This is not necessary in JPF.

A tool based on the Java Debugging API would have the following additional

shortcomings:

- Most standard Java VM's do not have a facility for forcing the garbage collector to run: System.gc() merely requests that the collector be run, but does not force it. Consequently, our prototype may observe some drag time between when an object becomes collectable and when it is actually collected. This is especially problematic for small example programs, where the collector will never be run.
- The Java debugging API on most standard virtual machines does not filter out class loading activity, and so this kind of system activity appears to the prototype as part of the application program.

5.2 Jikes

Jikes is a Java virtual machine written in Java. It is a research VM created by IBM.

Implementing our tool using the Jikes VM would have meant modifying the VM to report events relating to object creation and program flow. This would be easier than modifying a production VM, as the programming could be done in Java and because the VM is is built to be modified for experiments such as JunkAware.

Jikes also has limitations. For example, it does not support AWT. It also currently uses the GNU Classpath or Harmony class libraries and only has experimental support for the OpenJDK class libraries, limiting the programs that can be run under it.

5.3 Production VM

Most production Java virtual machines (such as Sun's) are at least partially written in C. These VMs are also highly optimized, and thus their implementations are likely not to be straightforward. These factors combined would have made modifying a production VM difficult.

Sun's VM, which is the most widely used JVM, was also not open source at the time this project was started, so obtaining the source to modify would have been difficult.

Chapter 6

Evaluation

We believe Object Lifetime Specifications allows us to express programs that would be difficult in other models. For example, we can express a multi-threaded producer/consumer pattern where we specify the object produced will be deallocated after the consumer has finished processing it. We have written such a program, annotated it with Object Lifetime Specifications, and checked it with JunkAware.

The program has three threads: `main`, `producer`, and `consumer`. The `main` thread constructs the other two threads, as well as the queue object that they share; once the two other threads are started, the `main` thread terminates. The `producer` thread creates data objects and puts them on the queue; the `consumer` thread dequeues the data objects and prints them out. Source code for this example is listed in Figures 6-1, 6-2, and 6-3.

The main property that we want to check is that the data objects are released after they are printed out — that they don't accumulate in an unbounded fashion. We can check this property with two annotations: `@Creates("data")` on the method in the producer that creates and enqueues the data, and `@Releases("data")` on the method in the consumer that dequeues and prints the data.

The interpretation of these annotations is as follows: Every time the the annotated

producer method executes, it creates a new group named data_i, where i is an index number that gets incremented for every execution of the method. The @Releases annotation needs to be combined with a quantifier, as can be seen in Figure 3-5. This quantifier may be quite general, *i.e.*, All or Some; or may describe if the shared queue is FIFO (Oldest) or LIFO (Newest). It's a FIFO queue, and so we choose the @ReleasesOldest("data") annotation.

We can enrich this example by thinking a little more deeply about the queue. There are a variety of ways that it may be implemented, including an array, a linked list, or some kind of re-balancing structure (like a red-black tree or a hashtable). Some of these possible implementation strategies involve creating bucket objects to hold the data, and in the case of a re-balancing structure, potentially reorganizing these buckets and creating new buckets at times that are obvious neither to the producer nor to the consumer. We would like to maintain the modularity property that the producer and consumer only need to know about how they use the queue (*e.g.*, FIFO, LIFO, *etc.*), and do not need to know about how the queue is internally implemented.

The @Pins annotation gives us this modularity. The queue can pin its buckets, and they can even pin each other. What this means is that the group memberships of the queue are passed on to its buckets: *i.e.*, the buckets do not need to expire until the queue does. The alternative is that the expected lifetime of the buckets gets tied to the expected lifetime of some data object put in the queue, which would likely cause spurious specification violations — especially in the case of a re-balancing data structure.

Our approach with the @Pins annotation would allow the bucket objects to accumulate inside the queue in an unbounded manner. We argue that if the author of the queue data structure wishes to maintain a modularity boundary, then it is their responsibility to ensure that these kinds of leaks do not occur. There are a number

of techniques that can be employed to this end, including: using our annotations in a manner informed by the internal structure and workings of the queue; ordinary class invariants; and other special techniques developed specifically for finding memory leaks within data structures [11]. When one chooses to erect a modularity boundary, then one needs to take responsibility for what happens on the inside of that boundary.

Lines 17–20 of Figure 6-1 check to see if the "leak" flag as been passed on the command line. If it has, then the consumer will store the data it receives from the queue in a field in the consumer thread object. This will cause the data object to become junk because the the `consume()` has a `@Releases` annotation, meaning that the data object should have been released by the end of the method call.

When run under JunkAware without the "leak" flag, JunkAware correctly reports that the program conforms to its specification. When run with the "leak" flag, JunkAware correctly reports that the data object becomes junk.

For a single thread interleaving, running the program under JunkAware versus the Sun JVM seems to incur a factor of 2 slowdown. Requesting all thread interleavings seems to incur a factor of 100 slowdown versus a regular execution on the Sun JVM.

Figure 6-1: Main method of Producer/Consumer example

```
10 public class Main {
11
12     private static boolean LEAK = false;
13
14     public static void main(String[] args) {
15         System.out.println("starting prodcons ...");
16
17         // should we have a leak?
18         if (args.length > 0 && args[0].equals("--leak")) {
19             LEAK = true;
20         }
21
22         // actual work
23         final int capacity = 1;
24         final BlockingQueue<Nugget> q = new LinkedBlockingQueue<Nugget>(capacity);
25         final Producer p = new Producer(q);
26         final Consumer c = new Consumer(q);
27         c.start();
28         p.start();
29
30     }
31
32     static final class Nugget {
33         final int value;
34         Nugget(final int value) { this.value = value; }
35         @Override public String toString() { return Integer.toString(value); }
36         @Override public int hashCode() { return value; }
37         @Override public boolean equals(final Object other) {
38             if (other instanceof Nugget) {
39                 return value == ((Nugget)other).value;
40             }
41             return false;
42         }
43     }
```

Figure 6-2: Producer from the Producer/Consumer example

```
45    static class Producer extends Thread {
46        final BlockingQueue<Nugget> q;
47
48        Producer(final BlockingQueue<Nugget> q) {
49            super("Producer");
50            this.q = q;
51        }
52
53        @CreatesRegion("Producer")
54        public void run() {
55            // real data
56            for (int i = 0; i < 1; i++) {
57                produce(i);
58            }
59            // poison pill to end the program
60            produce(-1);
61            System.out.println("Producer done.");
62        }
63
64        @CreatesRegion("nugget")
65        private void produce(int i) {
66            try {
67                q.put(new Nugget(i));
68            } catch (InterruptedException e) {
69                e.printStackTrace();
70            }
71        }
72    }
```

Figure 6-3: Producer from the Producer/Consumer example

```
74      static class Consumer extends Thread {
75          final BlockingQueue<Nugget> q;
76          Object leak = null;
77
78          Consumer(final BlockingQueue<Nugget> q) {
79              super("Consumer");
80              this.q = q;
81          }
82
83          @ReleasesSome("Producer")
84          public void run() {
85              while (consume()) {}
86              System.out.println("Consumer␣done.");
87          }
88
89          @ReleasesOldest("nugget")
90          private boolean consume() {
91              try {
92                  final Nugget s = q.take();
93                  System.out.println("Consumer:␣␣" + s + "␣...␣" + leak);
94                  if (LEAK == true && s != null && leak == null) {
95                      leak = s;
96                  }
97                  if (s == null) return false;
98                  if (s.value == -1) return false;
99                  return true;
100             } catch (InterruptedException e) {
101                 e.printStackTrace();
102             }
103             return false;
104         }
105     }
106 }
```

Chapter 7

Conclusion

Memory leaks in Java programs and other languages with garbage collection arise because the programmer has retained references to objects that are no longer needed. Such objects, called junk, are both difficult to detect and to reason about. They are difficult to detect because they are the objects that are not active in the computation. They are difficult to reason about because objects can be shared and thus there is no obvious natural hierarchical structure among them.

Object Lifetime Specifications help the programmer detect and reason about junk objects by tying the end-time of objects to the naturally occurring hierarchical structure of programs, threads, and stack frames.

We have implemented a runtime verifier for Object Lifetime Specifications called JunkAware using the Java Path Finder virtual machine. Given a user program annotated with Object Lifetime Specificationss for analysis, JunkAware will read the specifications and execute the program. If the execution terminates, then JunkAware will report a definitive judgment as to whether there were junk objects in that execution. Running the user program under JunkAware versus the Sun JVM seems to incur about a factor of 2 slowdown.

Even if one execution does not contain junk objects, there may be others that

do. Therefore, to increase the effectiveness of the verification, JunkAware uses JPF's backtracking facility to explore multiple executions that are related by some non-deterministic choice (such as thread interleavings). For our simple producer-consumer example program, the slowdown for searching all thread interleavings seems to be about a factor of 100.

Although more evaluation and analysis of Object Lifetime Specifications is required to determine their effectiveness in real-world programs, JunkAware provides the opportunity to do this future work.

Bibliography

[1] Benjamin Brosgol, James Gosling, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java.* Addison-Wesley, Reading, Mass., 2000. ISBN 978-0201703238. URL `http://www.rtsj.org/specjavadoc/book_index.html`.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, Mass., 1995.

[3] R. Hastings and B. Joyce. Purify: A Tool for Detecting Memory Leaks and Access Errors in C and C++ Programs. In *Proceedings of the Winter 1992 USENIX Conference*, pages 125–138, Berkley, CA, January 1992.

[4] Fritz Henglein, Henning Makholm, and Henning Niss. A direct approach to control-flow sensitive region-based memory management. In *Proc.3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 175–186, Montréal, Canada, 2001. ACM.

[5] Maria Jump and Kathryn S. McKinley. Cork: Dynamic memory leak detection for garbage-collected languages. In Matthias Felleisen, editor, *34th POPL*, pages 31–38, Nice, France, January 2007.

[6] Nick Mitchell and Gary Sevitsky. LeakBot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In Luca

CARDELLI, editor, *Proc.17th ECOOP*, volume 2743 of *LNCS*, Darmstadt, Germany, July 2003. Springer-Verlag. ISBN 3-540-40531-3.

[7] NICHOLAS NETHERCOTE AND JULIAN SEWARD. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In JEANNE FERRANTE AND KATHRYN S MCKINLEY, editors, *Proc.PLDI*, San Diego, CA, June 2007.

[8] NIKLAS RÖJEMO AND COLIN RUNCIMAN. Lag, drag, void and use — heap profiling and space-efficient compilation revisited. In RICHARD L. WEXELBLAT, editor, *Proc.1st ICFP*, pages 34–41, Philadelphia, PA, May 1996. ACM Press, NYC, NY. ISBN 0-89791-770-7.

[9] DEREK RAYSIDE AND LUCY MENDEL. Object ownership profiling: A technique for finding and fixing memory leaks. In ALEXANDER EGYED AND BERND FISCHER, editors, *Proc.22nd ASE*, Atlanta, GA, November 2007.

[10] JULIAN SEWARD AND NICHOLAS NETHERCOTE. Using Valgrind to detect undefined value errors with bit-precision. In *Proc.USENIX*, April 2005.

[11] RAN SHAHAM, ELLIOT K. KOLODNER, AND MOOLY SAGIV. Automatic removal of array memory leaks in Java. In DAVID A. WATT, editor, *Proc.9th CC*, volume 1781 of *LNCS*, Berlin, March 2000. Springer-Verlag. ISBN 3-540-67263-X.

[12] BRUCE A. TATE. *Bitter Java*. Manning, 2002. ISBN 193011043X.

[13] WILLEM VISSER, KLAUS HAVELUND, G. BRAT, S. PARK, AND F. LERDA. Model checking programs. *Automated Software Engineering Journal*, 10(2), 2003.

[14] PHILIP WADLER. Linear types can change the world! In M. BROY AND C. JONES, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 347–359. North Holland, 1990.