The Big Manual for the Java String Analyzer

Asger Feldthaus and Anders Møller
Aarhus University
{asf,amoeller}@cs.au.dk

November 30, 2009

Contents

Chapter 1

Overview

The Java String Analyzer, called JSA, is a tool for performing static analysis of Java programs. Its purpose is to predict the possible values of string expressions in an attempt to validate the correctness of a program. The analysis is based on the technique described by Christensen, Møller and Schwartzbach [1]. The JSA home page at

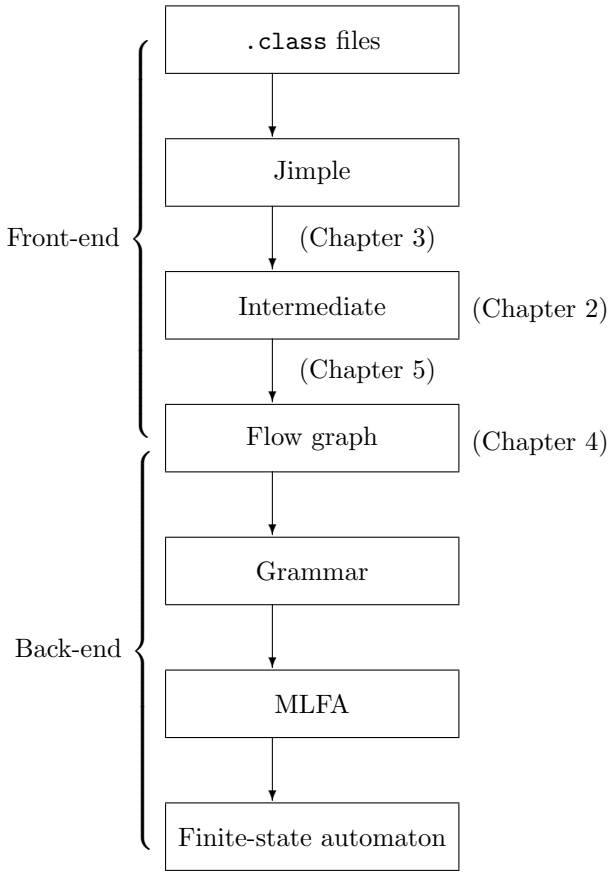
`http://www.brics.dk/JSA/`

provides a more detailed reference of the classes than found in this manual, including Javadoc documentation. This manual describes version 2.1 of JSA. It is intended both for developers of tools that build on top of JSA and for those extending JSA with new functionality.

As input, the tool takes the `.class` files to analyze, which we call the *application classes*. In the application classes, one or more string expressions are selected as *hotspots*. For each hotspot, the tool produces a finite-state automaton whose language contains all possible Unicode strings that the hotspot might evaluate to at runtime. This is a sound approximation: the output automaton may accept more strings than may actually occur, but not the opposite.

1.1 Phases

The JSA tool consists of several phases, each transforming the program into a new form. The phases are separated into two blocks: The *front-end* and the *back-end*. They are all outlined here:



The front-end is the focus of this manual; the back-end is best described in the research paper [1] and the Javadoc. The first phase which converts `.class` files to Jimple code is implemented by the Soot framework. For help with Soot, see to the Soot survivor's guide [2]. Automata are represented using the `dk.brics.automaton` package [3].

1.2 Interaction with External Code

The analyzed program may interact with classes not part of the application classes – we call such classes *non-application classes*. We consider these three types of interactions between application and non-application code:

- Application code calls a non-application method.
- Non-application code calls an application method.
- Non-application code assigns to an application field.

All three can be moderated by client-defined *resolvers* and *external visibility*, which provide information about the boundaries between non-application and application code. Only externally visible methods and fields are assumed to be usable from non-application code. Calls to non-application methods can be handled precisely by a resolver, as it may specify possible return values, and which mutable arguments might be modified externally. Chapter 6 discusses how to implement resolvers and external visibility strategies.

By default, all public methods and fields are externally visible. In the rare event that protected methods and fields are used outside the application, another external visibility must be used.

1.3 Customization

Other languages than Java can be analyzed if one can translate the other language to either the intermediate format or the flow graph format. The analysis can then continue from there on; neither format is connected to the Java language or its class library (however new automata operations may be necessary).

To hand-tailor the analysis to one's own application or framework, resolvers and an external visibility strategy can be implemented as Java classes and used with the analysis. One can also choose to stop the analysis after the grammar has been created and output that, instead of extracting a finite-state automaton for each hotspot.

1.4 Unsoundness

Despite the goal of analysis soundness, unsoundness may occur if an application class is subclassed by a non-application class interacting with the application. This includes application interfaces implemented by non-application classes and dynamic classes such as those created by `java.lang.reflect.Proxy`.

The analysis is designed to work with the whole program, in which case such external subclassing should not occur.

1.5 Using JSA

There are two ways to analyze an application using JSA:

- Write a program specifying which expressions to use as hotspots and their expected languages. One can also provide custom resolvers and external visibility. This is described in Chapter 6
- Use the runtime system. In the program being analyzed one must insert method calls and annotations to mark which expressions are to be analyzed. This is described in Chapter 7.

Chapter 2

Intermediate Format

`package dk.brics.string.intermediate;`

The *intermediate format* is an abstract program representation, implemented in the package `dk.brics.string.intermediate`. A program in this format is called an *intermediate program*. It represents the program being analyzed, but with uninteresting details removed. It has no textual representation, and only exists as objects in memory.

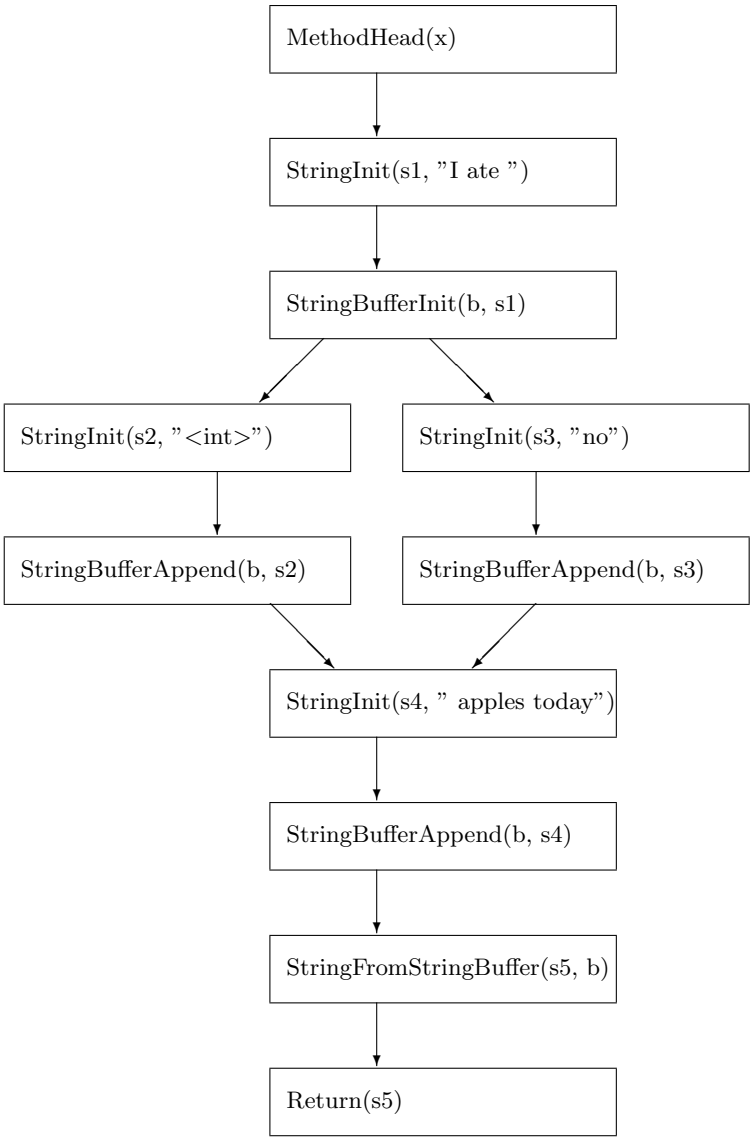
Unlike Jimple and Java bytecode, the intermediate form is abstract and not intended for concrete execution.

The central classes are `Application`, `Method`, and `Statement`. These are arranged in a tree-like fashion: An application contains methods, and a method contains statements. Statements are arranged in a control flow graph. Creating a method requires an already existing application object, and creating a statement requires an existing method object. This allows for strong invariant enforcement and unique key-indexing.

Below is an example Java method, followed a corresponding intermediate method.

```
public String foo(int x) {
    StringBuffer b = new StringBuffer("I ate ");
    if (x > 0) {
        b.append(x);
    } else {
        b.append("no");
    }
    b.append(" apples today");
    return b.toString();
}
```

The corresponding intermediate program is a graph:



Above, the symbol "<int>" denotes the finite-state automaton accepting integer strings such as "`17`" or "`-3`", and the other strings represent an automaton accepting only that string. Illustrating the program as a control flow graph consumes a good amount of space, so from now on we will instead write intermediate programs in pseudecode.

2.1 Intermediate Statements

Below is a complete list of all the statement types in intermediate programs:

`ArrayAddAll`	`Catch`	`StringBufferAppend`
`ArrayAssignment`	`ExceptionalReturn`	`StringBufferAppendChar`
`ArrayCorrupt`	`FieldAssignment`	`StringBufferAssignment`
`ArrayFromArray`	`FieldReference`	`StringBufferBinaryOp`
`ArrayNew`	`Hotspot`	`StringBufferCorrupt`
`ArrayStatement`	`MethodHead`	`StringBufferInit`
`ArrayWriteArray`	`Nop`	`StringBufferPrepend`
`ArrayWriteElement`	`ObjectAssignment`	`StringBufferUnaryOp`
`AssertAliases`	`ObjectCorrupt`	`StringConcat`
`AssertBinaryOp`	`PrimitiveAssignment`	`StringFromArray`
`AssertUnaryOp`	`PrimitiveFromArray`	`StringFromStringBuffer`
`BasicBinaryOp`	`PrimitiveInit`	`StringInit`
`BasicUnaryOp`	`Return`	
`Call`	`StringAssignment`	

Most statements have one or two references to the intermediate variables that they read from and/or write to. `StringInit`, for example, has one intermediate variable and a finite-state automaton. It intuitively assigns an arbitrary string from the automaton's language to the variable. `StringFromArray` has two intermediate variables, `from` and `to`, and assigns an arbitrary string from the string array `from` to the string variable `to`. `Hotspot` takes one intermediate string variable, and does nothing except mark the presence of a hotspot. `MethodHead` is always the first statement of a method, and `Call` and `Return` work as method calls normally do in an imperative programming language. See the Javadoc for more detailed descriptions of the statements.

2.2 Mutual References

There are several *two-way* references in the intermediate form. For example, a statement lists both its predecessors and successors. It is guaranteed that each successor of a statement S will list S as one of its predecessors. These properties are *unbreakable* in that careless use of the API cannot break them. More examples of guaranteed mutual references are:

- `method.getApplication()` always returns the application that contains `method`.
- `statement.getMethod()` always returns the method that contains `statement`.
- `method.getCallSites()` always returns all `Call` statements that call `method`.
- `method.getReturns()` always returns all `Return` statements in `method`.

- `field.getReferences()` always returns all `FieldReference` statements referring to `field`.

If extensions are made to JSA, unbreakable invariants like these are preferred whenever possible.

2.3 Keys

Each variable, statement, and method is assigned a unique *key*. Keys are non-negative integers, dispensed in sequential order starting with number 0. Keys are unique only within their own family of objects; for example, two statements cannot have the same key, but a statement and a method might.

Keys are distributed by the `Application` object. Methods and variables receive their key when constructed, while a statement first receives it when it gets added to a method.

The keys provide the perfect means for efficient bitmap and bitset implementations, though such techniques are not currently implemented. Keys also provide a consistent way for these classes to implement the `Comparable` interface, which orders them by their key.

2.4 Variable Types

An intermediate program is statically typed, as each variable has a type denoting which type of objects it might contain. The intermediate types are named after the Java-type counterparts but can more generally be thought of as *immutable* strings, *mutable* strings, *collections* of strings, and *characters* (the primitive type).

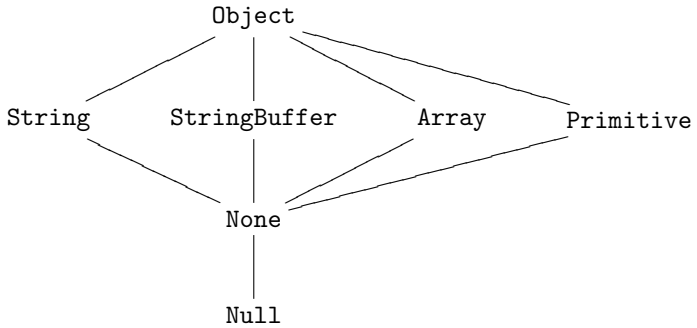


Figure 2.1: Lattice of intermediate variable types.

The type `None` indicates a type of object we are not interested in, and `Null` indicates a variable that can only hold the value `null`. `StringBuffer` is used to model the Java types `java.util.StringBuffer` and `java.util.StringBuilder`.

`Array` models arrays of any type and dimension and `java.util.Collection`. Note that variables of type `StringBuffer` and `Array` are *references* to such objects, so there may be aliasing between mutable variables. We therefore sometimes talk about a mutable *object*, since more than one variable may refer to it.

When Java assertions are enabled, the constructors for each intermediate statement will assert that the variables involved are not of incompatible types. For example, attempting to create a `StringBufferAppend` statement with an `Array` variable will fail.

Primitive types are modeled as strings of length one. Booleans are also treated as primitive types, where their only possible values are the Unicode values 0 and 1.

Variable types are currently only used to assist the translation to and from the intermediate format; thus the precision would be the same if every variable were assigned the type `Object` before converting it to a flow graph. However, they are exceptionally good at validating the correctness of our translation.

2.5 Variables and Fields

There exist two types of variables: *local* variables and *field* variables. A local variable has scope only within one method, while a field variable can be accessed from any method. Field variables may only be directly modified by the statements `FieldReference` and `FieldAssignment`. To modify a mutable field variable, it must be moved to a local variable, and the local variable should then be modified. For example (here using an imaginary syntax for intermediate code):

```
// Wrong! Fields cannot be modified directly.
s = "foo"
field.append(s)

// Correct! Fields may be modified through aliases.
s = "foo"
b = field
b.append(s)
```

Like with variable types, when assertions are enabled, the statement constructors will enforce this requirement.

2.6 Parameters and Parameter Aliases

Each method has two sets of variables representing its parameters: the *parameters* and the *parameter aliases*. They are initially identical, but if a parameter is reassigned, only the *alias* variant will get a new value. The reason has to do with the alias analysis discussed in Section 5.5.3.

2.7 Corruption

A mutable object may become *corrupt*, after which reading its value yields a completely arbitrary string. Corrupt objects may spontaneously change value, and modifications made to a corrupt object have no effect. Corruption exists to accommodate for the unknown modifications made by non-application classes and client-defined collection types (see Section 3.5). If a mutable object escapes into unknown code, it is corrupted unless one of the resolvers says it should not be (see Chapter 6 about resolvers).

2.8 Exceptional Flow

The statements `Catch` and `ExceptionalReturn` exist to handle exceptional control flow in an intermediate program. Each intermediate method has one exceptional return statement. A control flow edge ending in either type of statement represents exceptional flow, and is therefore called an *exceptional edge*. There is also an implicit exceptional edge from an `ExceptionalReturn` statement to every `Call` statement calling its method.

At a statement with an outgoing exceptional edge, an exception may be thrown *after* the statement completes – or in case of a `Call` statement, it may be thrown *during* the call. If an exception can be thrown *before* the statement has any effect, the exceptional edge is placed at the statement's predecessors. When an exception is thrown, control flow may continue at any of the `Catch` statements reachable by following exceptional edges, including the implicit ones.

Note that `Catch` statements represent an exception that was successfully caught, so they should not have outgoing exceptional edges to accommodate for uncaught exceptions. If the thrown exception might not be caught by a particular exception handler, an additional edge should be added from the failing statement to the `ExceptionalReturn` or similar enclosing exception handler.

2.9 Assert Statements

The intermediate program supports path sensitivity through *assert* statements.

Each assertion refers to a *target statement*, and asserts that some condition was true after the target statement was last executed. Their condition therefore refers to some past state of the program. We say that an assertion is *valid* if its condition also holds when the assertion itself is reached. Assertions without that property are called *invalid*, though they are allowed to exist. The concept of valid and invalid assertions should not be confused with *unsoundness*, as the analysis will simply ignore invalid assertions. The reason invalid assertions exist is because we do not always know in advance whether an assertion is valid, thus we allow them to be created and then detected later in the analysis.

To illustrate, here is a piece of pseudo code pointing out the location where a valid assertion might be created:

```
b = x.contains(y);
if (b) {
     assert x.contains(y);
}
```

In the following code, however, that assertion would be invalid:

```
b = x.contains(y);
x = "foo";
if (b) {
     assert x.contains(y); (invalid)
}
```

In both cases, the assignment to `b` is the target statement of the assertion, because we only really know that the condition was true at that time in the program. The invalid assertion is allowed to exist in the intermediate program, but it will be detected by the analysis described in Section 5.7. The motivation for this design is to simplify creation of assertions, described in Section 3.4. Note that an assertion's reference to a target statement is only used to detect its validity - after the validity test the reference is no longer used.

There are three types of assert statements. In addition to its target statement each assert statement has the parameters described below.

- `AssertAlias(Variable a, Variable b, boolean x)`
  Asserts that `a` and `b` are aliases if and only if `x` is true.
- `AssertUnaryOp(Variable a, UnaryOp op)`
  Asserts something about `a`, specified by the given unary operation. Examples of such unary operations are `AssertEmpty` and `AssertHasLength`.
- `AssertBinaryOp(Variable a, Variable b, BinaryOp op)`
  Asserts something about `a` specified by the given binary operation. Examples of binary operations are `AssertEquals`, `AssertContains` and `AssertContainedIn`. These assertion usually come in pairs, so one can assert something about `b` as well.

The last two types are called *operation* assertions (because they use string operations). String operations operate on finite-state automaton during the final stages of the analysis. The operation used in an operation assertion should produce a subset of the input variable's language, or the assertion might actually decrease the precision of the analysis.

It is always sound to ignore an assertion or remove it from an intermediate program.

2.10 Semantics

We of course never execute an intermediate program, but having some semantics for it is useful in order to assert correctness of a translation to or from it.

An intermediate program generally does not have sufficient information to execute deterministically. Examples of nondeterminism are choosing which successor statement to execute next, or which string to read from an array. Given all the necessary nondeterministic choices, we can actually think of an intermediate program as being executable. To argue whether an intermediate program is a correct model for a given program, we use the following somewhat short and informal semantics:

An intermediate program I *models* another program P if for every possible execution of P there exists a set of nondeterministic choices so that I has a corresponding execution. Two executions are *corresponding* if every string hotspot evaluates to the same string in the two executions, and no assert statements fail in the intermediate program's execution. An assertion is said to *fail* if it is reached but its condition did not actually hold at the time the target statement was executed. The correspondence between hotspots in the input program and the intermediate program is not specified here.

Chapter 3

Jimple→Intermediate

```
package dk.brics.string.java;
```

This phase translates a Jimple program to an intermediate program. The root of the translation takes place in `Jimple2Interface`, though many tasks are delegated to other classes to keep the code structured, as seen in the next section.

3.1 Translation of Methods Bodies

Figure 3.1 displays how the translation of method bodies are delegated into successively smaller tasks. All method calls are declared in an interface without the `Impl` suffix to make the dependencies very explicit. In the following, we outline the tasks performed by each of the main classes.

```
Jimple2Intermediate
    ↓
MethodTranslatorImpl → AssertionCreatorImpl
    ↓
StatementTranslatorFacadeImpl
    ↓
StatementTranslatorImpl
    ↓
CompositeMethodCallTranslator
    → BuiltInMethodCallTranslator
    → ResolverMethodCallTranslator
    → ApplicationMethodCallTranslator
```

Figure 3.1: Delegation of tasks in the translation of a method body.

Assertion creation is described in Section 3.4. The other tasks will be described below.

3.1.1 Method Translator

The `MethodTranslatorImpl` strategy runs a control flow analysis of the Jimple method, considering both normal and exceptional flow. A `Catch` statement is created for each `catch` block (`finally` blocks do not exist in bytecode). Each statement is translated into an isolated control flow graph with unique head and tail statements. These subgraphs are then connected by heads and tails using the information from the normal control flow, and exceptional edges are added to all relevant `Catch` statements and the `ExceptionalReturn` statement. The graph is finally placed after the `MethodHead` statement to complete the method's control flow graph.

Assertions are created after all statements in a method's body have been translated to isolated graphs but before they are connected. The method translator searches for `if` and `switch` in the method and requests its `AssertionCreator` instance to create assertions for each such statement it finds. The assertion creator returns an isolated graph with assert statements (possibly empty) which the method translator must connect with the remaining statements.

This class also performs the nullness analysis described in Section 3.3.

3.1.2 Statement Translator Facade

`StatementTranslatorFacadeImpl` is a layer between the method translator and the statement translator. Translating a statement requires a lot of high-level information, such as which intermediate variable corresponds to a given Jimple variable. The facade must make all such information easily available to the statement translator to minimize the amount of code and complexity of that class. These methods are declared in the `IntermediateFactory` interface, also implemented by the facade.

The class also acts as a facade towards the method translator. One Jimple statement may be translated into several intermediate statements, and this class arranges them into a subgraph and returns the heads and tails to the method translator.

3.1.3 Statement Translator

As the name suggests, `StatementTranslatorImpl` translates a Jimple statement into intermediate statements. The result of such a translation is entirely defined by the side-effects imposed by the method calls it makes to the `IntermediateFactory` object it receives as parameter. The motivation for this design is to simplify this otherwise obese class.

The class implements a huge visitor pattern to handle all Jimple statements and expressions. Each visitor method for an expression creates zero or more statements and returns a variable holding the result of the expression. The visitor methods also receive as argument the Jimple `ValueBox` containing the expression, because a hotspot expression must remember which Jimple expression it originated from.

3.1.4 Method Call Translator

The method call translator, `CompositeMethodCallTranslator`, is the only strategy further decomposed into smaller classes. The class `BuiltInMethodCallTranslator` handles well-known method calls such as `StringBuffer.append` and creates statements modeling them very precisely. `ResolverMethodCallTranslator` asks the client's resolvers to provide a precise modeling of other methods. Last, `ApplicationMethodCallTranslator` creates a `Call` statement if the called method is in the application being analyzed. The precision of the entire analysis depends largely on `BuiltInMethodCallTranslator`.

If more than one translator could translate a given method call, the one with highest precedence is chosen (see the Javadoc for details). If none of the translators can provide a precise translation, then `StatementTranslatorImpl` will react conservatively by corrupting every variable involved.

3.2 The Wrapper Method

In `Jimple2Intermediate`, a special intermediate method named `<wrapper>` is created. This method calls every *externally visible* method with corrupt arguments, and assigns every externally visible field to a corrupt value. Before this, it also sets every `String`-type field to the string `"null"` (explained in Section 3.3).

3.3 Nullness Analysis

When predicting the value of an expression like `("x" + i)`, it is useful to know whether `i` can be `null` or not. If `i` has type `Integer`, for example, it is not guaranteed to produce an integer string if it could be `null`.

This analysis is provided by Soot's `NullnessAnalysis` class and is performed by the method translator. The results of the nullness analysis is passed to the statement translator facade, which makes the results available to the statement translator through the `IntermediateFactory` interface.

The built-in method call translator exploits the information. If `String.valueOf` is invoked with an argument that might be `null`, the control flow is split in two. One branch produces the `"null"` string, and the other models the `toString` method of the argument.

Soot's nullness analysis, however, is too imprecise for our needs. It does not realize that certain methods like `StringBuffer.toString()` never return `null`, which tends to have a heavy impact on precision. To endure this, we never treat `String` variables to have the value `null`, but rather the string-value `"null"`. This does introduce other imprecisions, but those are less critical.

3.4 Assertion Creation

The assertion creator uses a reaching definitions analysis provided by Soot to analyze the consequences of a given condition being true. As an example, consider the branch below:

```
if b = 0 then goto L
```

In this case we want to examine under which circumstances `b` might be 0 (the same as `false`). For each statement S that might have defined `b`, a separate branch of assertions is created. Suppose now that one of the possible definitions of `b` is (written in pseudo Jimple):

```
b = s1.equals(s2)
```

At this point, the following two assertions are created, both with the above statement as the target statement:

```
AssertBinaryOp(s1, s2, AssertEquals)
AssertBinaryOp(s2, s1, AssertEquals)
```

Suppose now that the program fragment that was analyzed looked like this:

```
b = s1.equals(s2)
s1 = "foo"
if b = 0 then goto L
```

Oops! It would appear that `s1` and `s2` may not actually be equal after the branch. The assertions we generated are *invalid*. This motivates the design we described back in Section 2.9 - we do not have to worry about our assertions being invalid. They will be removed for us later (see Section 5.7). This significantly simplifies the job of the assertion creator.

The above example put aside, the algorithm that creates assertions works by recursive application of these two functions, which return nothing but create assertions as side-effects:

```
void assertBoolean(Expr e, boolean expected)
void assertInteger(Expr e, int expected, Relation rel)
```

`assertBoolean` creates assertions that will hold when the specified expression evaluates to the `expected` boolean parameter.

A `Relation` denotes one of the six relations: $=, \neq, <, \leq, >, \geq$. `assertInteger` creates assertions that hold when the expression's result relates to the `expected` integer according to `rel`. For example, if `rel` is $<$, then it asserts that the expression evaluates to something strictly less than the `expected` parameter.

Here is how `assertBoolean` reacts to different types of expressions, written in pseudo-code:

- Local variable v:

```
split control flow here
for each assignment v = y that may have defined v
    start new control flow branch
    call assertBoolean(y, expected)
join every branch created above
```

- Binary *and* expression, a & b:

```
if expected == true
    call assertBoolean(a, true)
    call assertBoolean(b, true)
else
    split control flow here
    start new control flow branch
    call assertBoolean(a, false)
    start new control flow branch
    call assertBoolean(b, false)
    join the two branches
```

- Binary *or* expression, $a \mid b$:

```
if expected == false
    call assertBoolean(a, false)
    call assertBoolean(b, false)
else
    split control flow here
    start new control flow branch
    call assertBoolean(a, true)
    start new control flow branch
    call assertBoolean(b, true)
    join the two branches
```

- Boolean comparison $b == x$, where x is constant:

```
call assertBoolean(b, expected == (x == 1))
```

- Integer comparison, $i == x$, $i \leq x$, etc, where x is constant:

```
if expected == true
    call assertInteger(i, relation, x)
else
    call assertInteger(i, negate(relation), x)
```

- Method invocation `b.M(a0, a1, ...)`

```
if M is String.contains
    create AssertBinaryOp(b, a0, new AssertContains())
    create AssertBinaryOp(a0, b, new AssertContainedIn())
else if M is String.equals and a0 is a string
    ...
<long list of similar if statements>
```

This should hopefully show the general idea of the algorithm. `assertInteger` works in a similar fashion. To avoid an infinite recursion loop the recursion stops if a cycle is detected.

3.5 Collections and Arrays

Various complications must be addressed to correctly model arrays and collections of strings. Arrays may be covariant, so an array declared as `Object[]` may be a string array, and many methods such as `clone()` and `toArray()` return `Object[]` even though we may know the result is an array of strings. Collections are even worse, because generic arguments are invisible after compilation. For example, we cannot distinguish `List<String>` from `List<Integer>`. Additionally, clients may define their own collection types that violate the behaviour we might expect from a collection.

Both types are modeled as `Array` variables in intermediate code. By default, we assume an array or collection only contains strings. If an element that might not be a `String` is inserted then the entire array or collection gets corrupted. In this way, even if a collection was declared as `List<Object>` we can model it precisely if only strings were inserted in it. Note that inserting a `StringBuffer` in an array/collection corrupts both the array/collection and the string buffer.

To defend against client-defined collections we use a list of *trusted* collections. Invoking the constructor of an untrusted collection corrupts the collection being created. The trusted collections are `ArrayList`, `LinkedList`, `Vector`, `HashSet`, `LinkedHashSet`, and `TreeSet`. Methods like `Collections.unmodifiableList` and `Arrays.asList` are modeled as simple identity methods (returns their argument), and many methods like `Collections.sort` are modeled as `nop` operations, since we ignore the ordering of elements.

Because newly constructed arrays contain `null` in every entry, we insert `"null"` in the array whenever an array is allocated. Note that not all arrays have to contain `null`, such as an array returned by `Collection.toArray()`. Only those allocated in client code are forced to contain `null`.

We also model iterators (`java.util.Iterator`). The surprisingly simple way of doing this is to treat them as `Array` variables that are aliases of the collection they belong to. In fact, `iterator()` and `listIterator()` are modeled as returning the collection they are called on. Then a method like `ListIterator.add` is no different from `Collection.add`, and `Iterator.next()` only has to return a random element from the collection. In case concurrent modification does not throw an exception at runtime, the iterators also reflect changes made to their collection while iterating. Modeling iterators means the analysis has reasonable precision in `for` statements like:

```
for (String s :  list) {
    // do something
}
```

There is room for further improvement in this area. The operations that remove strings from an array or collection are currently not modeled very well. Examples of such methods are `Collection.clear` and `Arrays.fill`. Extending the intermediate language with an `ArrayClear` statement would be the best way to support this. No assertions are generated for collection methods like `Collection.contains`, and collections of characters are not modeled. Maps are currently not modeled at all.

Chapter 4

Flow Graph Format

```
package dk.brics.string.flow;
```

The flow graph contains various types of nodes connected with directed edges, where a node represents an expression and an edge represents possible flow of data. Nodes can have zero or more *use* points, which are places where it accepts incoming edges. A concatenation node, for example, has two use points, for the left- and right-hand side, respectively.

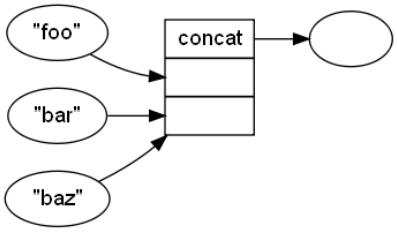


Figure 4.1: Example of a small flow graph. Possible values at the rightmost node are `foobar` and `foobaz`.

More complicated string operations are modeled using `UnaryNode` and `BinaryNode` objects with a `StringOperation` object. The back-end uses these objects to model their effect on finite-state automata.

The five types of nodes are:

```
AssignmentNode          ConcatenationNode       UnaryNode
BinaryNode              InitializationNode
```

For a description of the flow graph's semantics we refer to the paper [1].

Chapter 5

Intermediate→Flow Graph

`package dk.brics.string.intermediate.operations;`

The translation from an intermediate program to a flow graph requires the following analyses of the intermediate program, performed in the order listed:

1. field usage analysis
2. liveness analysis
3. detect invalid alias assertions
4. alias analysis
5. reaching definitions analysis
6. detect invalid operation assertions

5.1 Nop Statement Removal

`Nop` statements are convenient statements that do nothing when executed. Most `Nop` statements are removed before the intermediate program is converted to a flow graph, but in some circumstances a `Nop` statement will have to remain. This occurs when a `Nop` statement is the target statement of an assertion, *and* the `Nop` has multiple predecessors. If a `Nop` has only one predecessor, the assertion referring to it will have its target statement changed to the predecessor before the `Nop` is removed. A `Nop` that is not targeted will always be removed.

5.2 Field Usage Analysis

The field usage analysis analysis determines for each method which fields may be read from or assigned to as a result of invoking the method. If a method contains a reference or assignment to the field F, it is marked as *using* the field F. If a method M_1 contains a call to a method M_2 and M_2 uses the field F, then M_1 also uses the field F.

Note that a mutable field F may still be affected by an invocation to a method that does not use F if one its arguments is an alias for F. This is not an issue, however, since the call statement will redefine the value of its mutable arguments, and as such, the new value of the field will be defined there.

This analysis is only used to improve performance of the alias analysis.

5.3 Liveness Analysis

A variable is *live* at a program point if its current value might be read at a later point. The liveness analysis determines for each intermediate statement which variables are live before and after executing the statement.

Like the field usage analysis, the liveness analysis is only used to improve performance in the following alias analysis.

5.4 Detecting Invalid Alias Assertions

Recall from Section 2.9 that an *invalid* assertion is an assert statement whose condition might not be true when the assert statement itself is reached. Invalid alias assertions must be detected before the alias analysis, and the alias analysis is required by the reaching definitions analysis, which in turn is required to detect invalid operation assertions. For this reason, the two types of assertions must tested for validity at different times.

An alias assertion between variables a and b is invalid if some assignment to a or b may reach the assertion, but not its target statement. This analysis therefore performs a restricted type of reaching definitions analysis to detect such assertions.

Note that this is quite different from the following reaching definitions analysis, since side-effects are not considered relevant here. A statement like `b.append(x)` can be ignored because it does not change aliasing, while the other analysis will find it to be a definition of b.

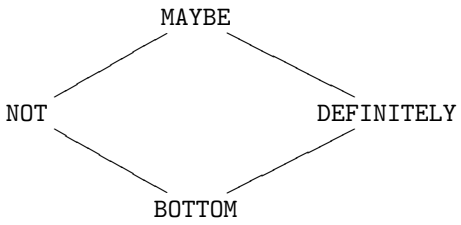
5.5 Alias Analysis

Two variables are called *aliases* if they refer to the same object. We additionally use the term *useful aliases* for variables referring to the same *mutable* object.

In our analysis, it is important to know which variables might be affected by a given string operation. Knowing which variables are aliases provides all the information required for this. Aliasing between immutable objects is of no interest, however, so we do not analyze their aliasing. Primitives have no aliasing so they are ignored as well.

The alias analysis is a simple, combined may/must alias analysis that determines for each program point the *aliasing status* between each pair of variables.

Aliasing status is a lattice represented by the `AliasStatus` enumeration:



The classes relevant for the alias analysis are `AliasAnalysis`, `AliasInfo`, `AliasTable` and `AliasStatus`. `AliasAnalysis` contains an `AliasInfo` object for each statement in the program, and each `AliasInfo` contains one `AliasTable` and a set of corrupt variables. The `AliasAnalysis` class searches for a fixed-point using the `WorkList` class and is responsible for interpreting each statement as an appropriate operation on the corresponding `AliasInfo` object.

Only *live* variables and *used* fields (as determined by the preliminary field usage and liveness analyses) are considered in the alias analysis.

5.5.1 Alias Table

Storing variable-pair-based aliasing information between a set of n variables corresponds to filling out an $n \times n$ matrix. For example, for a set of variables $\{a, b, c\}$, where a and b are aliases, the table storing definite aliasing could look like this:

	a	b	c
a	•	•	
b	•	•	
c			•

Observe now two important properties of any such aliasing table:

- The table is symmetric: If a is an alias for b, then b is also an alias for a.
- The diagonal is trivial: A variable is obviously an alias for itself.

Therefore, we only explicitly store the information required to reconstruct the entire table (an $\times$ marks a cell not stored in memory):

	a	b	c
a	$\times$	$\times$	$\times$
b	•	$\times$	$\times$
c			$\times$

An additional optimization takes advantage of variables types. Suppose now that a has type `Object`, b has type `StringBuffer`, and c has type `Array`. The

variables b and c can never be aliases, so it suffices to store an even smaller fraction of the table:

	a	b	c
a	×	×	×
b	•	×	×
c		×	×

Note that the table is in fact not filled with boolean cells as illustrated. Instead, each cells holds one of the four values represented by the `AliasStatus` enumeration.

5.5.2 Interprocedural Alias Flow

Aliasing information must be carefully propagated across method calls. If a method is called with two arguments that are aliases, the alias analysis of the body must this take into account. This aspect of the analysis is entirely handled in `AliasAnalysis` where it provides special treatment of `Call` and `Return` statements.

At a `Call` statement to method M, for each pair of arguments, their aliasing status is transferred to the corresponding pair of parameters in M's `MethodHead`. The aliasing status between arguments and fields are also transferred.

Likewise, `Return` statements transfer aliasing status between the return value, the arguments, and the used fields, as well as the corrupted status of the arguments (more on this in Section 5.5.3).

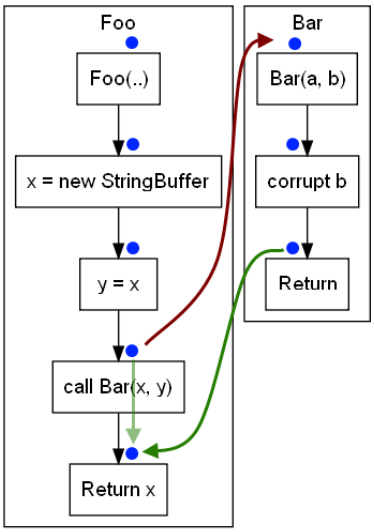


Figure 5.1: Interprocedural alias flow.

Figure 5.1 illustrates how interprocedural flow is handled. The dots represent

the `AliasInfo` objects between statements. An arrow from A to B indicates that aliasing information from A is used to update B. Aliasing from before the call statement is used to update the aliasing before the method head, and alias information from before the return statement *and* before the call statement is required to update the aliasing after the call statement.

The program in Figure 5.1 also demonstrates one of the nasty complications that arise in interprocedural aliasing. Observe that the x variable will be corrupt after the method call. When the analysis encounters the `Return` statement in `Bar`, it observes that the second parameter b is corrupt, and will therefore corrupt every alias of the second argument y at the call-site. However, y is not live after the call statement! Requesting its aliasing information there would be meaningless. It is crucial that it requests y's aliases from *before* the call-statement. x is indeed an alias for y before the call-statement (and both variables are live), so x will become corrupted after the call statement as it rightfully should.

When assertions are enabled, `AliasTable` intentionally throws an exception if requested aliasing status for non-live local variables or non-used field variables. This assertion is particularly good at detecting incorrect handling of programs like the one above.

5.5.3 Parameter Aliases

A method call must additionally determine which arguments were corrupted by the call. For example, consider this example intermediate program, written in pseudo code:

```
foo(x) {
    x = new StringBuffer;
    corrupt x;
    return;
}
bar() {
    a = new StringBuffer;
    call foo(a);
    // is 'a' now corrupt?
}
```

As the comment suggests, we want to know whether the argument is corrupted by the method call. In this case, however, the parameter x is reassigned before it is corrupted, so the original argument was not corrupted.

To handle this correctly, the parameter variables are always considered live, so at the return statement we can determine which arguments were corrupted.

5.6 Reaching Definitions

For every statement s and every local variable v being used by s, we want to know what statements could have assigned v its current value. This information is necessary to construct the edges in the flow graph. The `UsesVisitor` and `DefinesVisitor` classes provide the set of variables used and defined by a given statement. Here, the `DefinesVisitor` uses the results from the alias analysis to include aliases for modified variables.

The class `ReachingDefinitions`, which performs the analysis, does not use a fixed-point search. Each defined variable is being propagated along all successor paths until not live anymore or until killed by a strong definition.

5.7 Detecting Invalid Operation Assertions

An operation assertion is invalid if for any variable involved it has a reaching definition that is not also a reaching definition for its target statement. With the reaching definitions analysis already performed all there is to do is to test this for every operation assertion.

5.8 Flow Graph Creation

When the reaching definitions are found, the nodes of the flow graph are all created and then the edges are created last. Each statement produces exactly one node for each local variable it defines. These nodes are connected with edges according to the reaching definitions found earlier. Put simply, if there is a reaching definition from A to B, we create an edge from A's node to B's node.

Field variables are handled differently. Each field has one node, with one use point. Every statement that defines a field variable puts an edge from its own node to the field's node, and every statement that uses a field likewise adds an edge from the field's node to its own. This means only weak update is possible on fields, but the analysis is sound even for multithreaded applications.

Invalid operation assertions produce an assignment node instead of a `UnaryNode` or a `BinaryNode`. Corrupt variables always produce an `InitializationNode` with the *any string* language.

The classes handling this are `FlowGraphNodeCreator` and `FlowGraphEdgeCreator`.

5.8.1 Interprocedural Flow

A method call may modify the values of its mutable arguments, so not only the method's return value is interesting. For this reason, a `Return` statement defines all its method's mutable parameters, so a node is created for each one. A `Call` statement defines all its mutable arguments (and their aliases), and adds edges from the `Return` statement's nodes to the `Call` statement's nodes.

5.8.2 Exceptional Flow

The statements `Catch`, `ExceptionalReturn`, and `Call` require special attention. If a method modifies a mutable argument and then throws an exception before it completes all its modifications, the definitions from the called method's `Return` statement are not valid. For example, consider this pseudo code:

```
foo(StringBuffer b) {
    b.append("X");
    <maybe throw an exception here>
    b.append("Y");
}
bar() {
    b = new StringBuffer;
    try {
        foo(b);
    } catch {
        b.append("Z")
    }
    <hotspot: What is 'b'?>
}
```

The `Return` statement in `foo` will define `b` as definitely having `XY` appended, but it might instead have only `X` appended if an exception was thrown. Like the `Return` statement, the `ExceptionalReturn` statement defines every mutable parameter, so a `Catch` statement can add edges from them. The `Catch` statement in `bar` therefore observes that it has a `Call` statement as predecessor, and *weakly* defines `b` to take the value from `foo`'s `ExceptionalReturn` statement.

The `Catch` statement only defines mutable arguments to `Call` statements – modifications made directly in the `try` block are accounted for by the reaching definitions analysis, which does not distinguish normal and exceptional edges. This is why mutable arguments are only weakly defined by `Catch`; so they do not kill the definitions made directly in the `try` block.

Chapter 6

Running the Analysis from Java

`package dk.brics.string;`

The class stitching together all the phases is `StringAnalysis` in the `dk.brics.string` package. To use the class, one must prepare it using its static methods, and then instantiate it. The constructor will perform the whole analysis, and the results can then be queried using the resulting instance's non-static methods. This design prohibits that multiple applications are analyzed at once, which is unfortunate, but inevitable due to the way the underlying Soot framework works. An overview of the static methods are given here:

- `void addResolver(Resolver r)`
  Uses the specified resolver in the upcoming analysis (in addition to those already added).
- `SootClass loadClass(String name)`
  Loads the class with the specified name as an application class. The name must be given as a fully qualified classname, and the class must exist on Soot's classpath. Soot's classpath equals Java's active classpath by default but can be modified with Soot's `Scene` class. Returns Soot's representation of the class.
- `int addDirectory(String name)`
  Loads all classes found in the specified directory or jar file as application classes. If a directory is given, it must refer to root package. The directory or jar file is added to Soot's classpath before the classes are loaded. It returns the number of classes loaded.
- `void reset()`
  Removes all resolvers and application classes, so another application can be analyzed.
- `List<ValueBox> getExps(String sig, int argnum)`
  Returns a list of expressions occurring as argument to the specified method,

where `argnum` specifies which argument to get. This is useful for selecting hotspots in the application. Refer to the Javadoc for a description of `sig`'s syntax.

The constructor takes a collection of expressions to use a hotspots and optional external visibility strategy. When the analysis is complete, the automaton for a given hotspot can get acquired using the `getAutomaton(ValueBox)` method, where the `ValueBox` is one of the hotspot expressions.

6.1 Resolvers

Whenever a call to a non-application method is encountered, the analysis uses its *resolvers* to determine the effects of the call. A resolver may specify what is returned by the call and which of its mutable arguments might be modified by non-application code. Resolvers can also specify the possible values of fields accessed on non-application classes.

The `dk.brics.string.external.Resolver` interface declares two methods: `resolveMethod` and `resolveField`. Either method may return `null` to specify that the resolver knows nothing about the specified method or field, or it may return an instance of `MethodResolution` or `FieldResolution` if it does know something. The Javadoc contains the specifics of how to use these classes.

The analysis can use more than one resolver, and they are queried in the order they were added. The first resolver to give a non-`null` answer determines the resolution to use. If no resolver can resolve a particular method call or field reference (as is often the case), the analysis uses a sound worst-case approximation.

Clients can implement their own resolvers to increase the precision of the analysis, but the analysis only remains sound as long as the resolvers used provide sound answers. It is always sound for a resolver to return `null`.

6.2 External Visibility

It is possible that non-application code calls an application method or modifies an application field. The analysis accommodates for this using an *external visibility* strategy, which specifies which method and fields are visible to non-application code.

The `dk.brics.string.external.ExternalVisibility` interface declares two methods: `isExternallyVisibleMethod` and `isExternallyVisibleField`. Each returns a boolean denoting whether non-application code can access the method or field. Unlike resolvers, the analysis requires exactly one external visibility strategy.

The default external visibility assumes that public methods and fields are visible and non-public methods and fields are not. Technically, protected and package-private methods and fields *might* be accessed from non-application

code, but it was deemed too unlikely and too critical for precision to let the default strategy accommodate for that case.

Like resolvers, clients can implement their own external visibility strategy, but again, the analysis is only sound as long as the external visibility strategy provides sound answers. It is always sound for an external visibility strategy to return `true`. Providing an application-specific external visibility strategy can drastically increase the precision of the analysis.

Chapter 7

The Runtime System

`package dk.brics.string.runtime;`

To analyze a program, one can use JSA as a standalone tool and let the program itself describe how it should be analyzed. The program being analyzed should contain type annotations and methods calls to the static methods in the class `dk.brics.string.runtime.Strings`. These methods are:

- `String cast(String s, String regexp)`
  At runtime, the string must match specified regular expression, or an exception is thrown. If it does match, the same string is returned. The analysis will therefore assume the returned string is in the language of the regular expression. This can be used to assist the analysis "by hand" if it was found too imprecise, or simply as a runtime check.

- `boolean check(String s, String regexp)`
  At runtime, returns whether the string matches the specified regular expression. This does not affect the precision of the analysis, but the analysis will issue a warning if it detects that the check is always satisfied or never satisfied (indicating dead or redundant code).

- `String analyze(String s, String regexp)`
  At runtime, this does nothing except return the given string. It instructs the analysis to verify that the first argument does indeed always match the specified regular expression. For every analyze call in the program, the analysis will report either that it is always satisfied, or an example of a violating string that might occur. The analysis will use the term "exact match" if it concludes that the language of possible strings is exactly the language specified by the regular expression.

- `void bind(String s, String regexp)`
  Creates an identifier for the specified regular expression, so it can be used inside other regular expressions enclosed in sharp brackets (<,>). See the example in Section 7.4.

Each method taking a regular expression is overloaded and can also take a URL to a serialized automaton instead.

7.1 Annotations

The type annotation `@Type(String regexp)` may be used as an extension of Java's static type system. The annotated parameter, return type, or field must only be assigned strings matching the specified regular expression. Only `String` types may be annotated - string buffers, arrays, and primitives are not allowed. Aliasing between variables with different annotated types would complicate the type system too much. Chars might become annotatable in the future, but for now they are not.

When reading from an annotated string type, the analysis will always assume any string in the specified language may occur, even if it could derive a more precise set of strings.

Hotspots are automatically created to verify that one does not assign non-matching strings to such an annotated type.

The runtime system will abort with an error message if the following inheritance restrictions are not followed. If a method M_1 overrides or implements M_2, its return type must be more restrictive than or equal to that of M_2 and its parameter types must be less restrictive or equal. Unannotated return types and parameters inherit the annotation from their ancestor. If multiple methods are implemented, the return type becomes their intersection, and the parameter type becomes their union. It is forbidden to annotate a parameter of M_1 if the corresponding parameter in M_2 is not annotated.

Local variables cannot be annotated because such annotations are lost during compilation. The `@Type` annotation will therefore not allow itself to target a local variable.

The regular expression used in `@Type` may contain identifiers defined by `Strings.bind`, and it is strongly recommended to make use of that to reduce the amount of typing required, and to make refactorings easier. Alternatively, the annotation `@LoadType` works like `@Type`, except its argument is an URL referring to a serialized automaton defining the type's language.

If every method and field is fully annotated then the analysis is completely modular and each method can be analyzed independently of the rest of the program. If no annotations are used, the analysis becomes a *whole program analysis*. Using a moderate amount of annotations puts the analysis somewhere between the two extremes. A modular analysis generally makes it easier to find the source of an error or imprecision as the location of the reported problem will be much closer to the code actually causing the problem.

7.2 Running The Analyzer

To analyze an application using the runtime system, run the class `dk.brics.string.AnalyzeRuntime`. It accepts the following types of arguments, in any order. Unless otherwise specified, each type of argument may be used more than once (or not at all):

- `-dir=`*directory or jar*
  Use all class files found in the specified jar file or directory (and its subdirectories, recursively) as application classes. The directory must refer to the root package. The directory or jar file does not have to be on the classpath.

- *classname*
  Use the class with the specified fully qualified name or filename as an application class. The class must be on the classpath.

- `-resolver=`*classname*
  Use the class with the specified fully qualified name or filename as a resolver in the analysis. The class must be on the classpath. It must have a constructor taking no arguments and it must implement the `Resolver` interface. Resolvers are described in Chapter 6.

- `-externalvisibility=`*classname or identifier*
  Use the specified external visibility during the analysis, instead of the default. The value can either be a classname or filename of a class implementing the `ExternalVisibility` interface, or one of the predefined identifiers `public` or `main`. `public` is the default strategy as described in Chapter 6, and `main` is a strategy that declares only main methods to be externally visible (and no fields). At most one external visibility may be specified. If a class is specified, it must be on the classpath. External visibility is described in Chapter 6.

Make sure that JSA and all its required jar files (`automaton.jar` and `soot.jar`) are on the classpath when running it.

7.3 Example Ant Target

Here is an example of an `ant` target that will analyze the application placed in the `bin` folder, assuming all the necessary `jar` files are placed in the `lib` folder.

```
<target name="analyze">
  <java classname="dk.brics.string.AnalyzeRuntime" fork="true">
    <classpath>
      <fileset dir="lib">
        <include name="**/*.jar"/>
      </fileset>
```

```
    </classpath>
    <arg value="-dir=bin" />
  </java>
</target>
```

7.4 Example Program

For example, given this Java program:

```
public String bar(int x) {
    String s = "I ate " + x + " apples today";
    Strings.analyze(s, "I ate [0-9]+ apples today");
    return s;
}
```

The runtime system will report:

```
Dissatisfied by:  "I ate -1 apples today"
```

The regular expression for canonical signed integer strings, `0|-?[1-9][0-9]*`, can be cumbersome to write over and over again, so one can bind *identifiers* to be available in regular expressions. Suppose we rewrite the above program, to use the `bind` call:

```
static {
    Strings.bind("int", "0|-?[1-9][0-9]*");
}
public String bar(int x) {
    String s = "I ate " + x + " apples today";
    Strings.analyze(s, "I ate <int> apples today");
    return s;
}
```

The analysis will now print:

```
Exact match!
```

Carefully choose your regular expressions regarding numbers. The expression we used above will reject integers with leading zeros, even though such strings may be perfectly legal for your purpose. When working with floats, remember that `NaN` and `Infinity` may occur, and the analysis will always assume that they can occur.

Alternatively, we could have written the above program like so:

```
static {
    Strings.bind("int", "0|-?[1-9][0-9]*");
}
@Type("I ate <int> apples today")
public String bar(int x) {
    return "I ate " + x + " apples today";
}
```

Chapter 8

Test Suite

The test suite is designed to encourage an efficient workflow during development. Each top-level class in the `testcases.standalone` package is treated as a separate application to be analyzed, which we shall denote a *test case*.

A test case must contain exactly one call to the `StringTest.analyze` method. The first argument becomes a hotspot, the other three arguments are regular expressions indicating the expected outcome of the test. See the Javadoc for the complete specification.

The package `testcases.tools` contains the classes required to run the test suite. It is designed to be run with the JUnit tool built into Eclipse, providing a neat GUI and the means to debug individual test-cases. Running `PrecisionTester` as a JUnit Test will run the entire test suite. The result of a test-case is indicated by its color:

- Green (success) means a perfectly sound and precise answer was produced.
- Blue (failure) means the answer was sound, but imprecise. While being worse than the green outcome, this is generally acceptable.
- Red (error) means the answer was unsound, or an exception was thrown. This result is unacceptable.

Note that assertions should be enabled while running the test suite. The `PrecisionTester` class will print a warning in the console if they are disabled. By default, Eclipse will *not* enable them for you.

8.1 Customizing the Test Suite

Customized analyzers can be implemented using the `AutoTester` class. To create a custom tester, a class must implement `TestExecutor` and be annotated with `@RunWith(AutoTester.class)`. For example:

```
@RunWith(AutoTester.class)
public class MyTester implements TestExecutor {
    public void close() {}
    public void initialize() {}
    public boolean shouldTestRun(TestCase test) {
        return true;
    }
    public void executeTest(TestCase test) throws Throwable {
        if (test.getMainClassName().length() > 12) {
            throw new AssertionError("Long name");
        }
    }
}
```

Running the above class as a JUnit test will show a test case for every class in the `testcases.standalone` package, and fail those whose name is longer than 12 characters. See the source code in `FieldTester` for a more useful example.

Bibliography

[1] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium, SAS '03*, volume 2694 of *LNCS*, pages 1–18. Springer-Verlag, June 2003. Available from `http://www.brics.dk/JSA/`.

[2] Árni Einarsson and Janus Dam Nielsen. A survivor's guide to Java program analysis with Soot, 2008. Available from `http://www.brics.dk/SootGuide/`.

[3] Anders Møller. dk.brics.automaton – finite-state automata and regular expressions for Java, 2008. `http://www.brics.dk/automaton/`.

Index